TÉCNICO LISBOA

Does Big Data Require Big Systems? A Study of Complex Versus Lean Distributed Data Processing Systems

Duarte Miguel Montes do Nascimento

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. Miguel Filipe Leitão Pardal
Dr. Miguel Coelho Ferreira

Examination Committee

Chairperson: Prof. José Carlos Martins Delgado
Supervisor: Prof. Miguel Filipe Leitão Pardal
Member of the Committee: Prof. João Carlos Serrenho Dias Pereira

November 2022

Dedicated to Maria, Carlos and Gaia.

Acknowledgments

To my supervisors, Professor Miguel Pardal and Miguel Ferreira, for the support, motivation and guidance, and without whom this project would not have been possible.

To Carlos Cruz, Buyanjargal Shirnen, and the Unicage teams of Portugal and Japan for the support, motivation, hard work, and for providing complimentary access to Unicage software.

To IBM for providing complimentary access to all the needed infrastructure. To Arlindo Dias for guiding us through the IBM Cloud.

To IST for the wonderful last six years of my academic journey.

And of course, to my family and friends for constantly reminding me that there is more joy to life than the joy of studying and working.

To all, my deep and sincere gratitude.

Resumo

O paradigma dos macrodados é caracterizado pela necessidade de reunir e processar conjuntos de dados com grandes volumetrias, que chegam aos sistemas com grandes velocidades, numa variedade de formatos. Os sistemas especializados de processamento de macrodados, como o Apache Hadoop, o Hive e o Spark, oferecem abstrações úteis para lidar com os macrodados em conjuntos de máquinas distribuídos, mas são sistemas complexos e combinam muitas bibliotecas em extensas árvores de dependências. Como alternativa, Unicage depende somente do sistema operativo e das bibliotecas de tempo de execução do C para criar sistemas mais simples de processamento de macrodados.

Este estudo compara os desempenhos das duas abordagens - sistemas complexos ou simples - através de medições para o carregamento e processamento de conjuntos de macrodados estruturados e não estruturados em conjuntos de máquinas na nuvem de computação da IBM. As volumetrias dos conjuntos de dados de entrada variaram de 64 GB a 8192 GB.

Os resultados experimentais mostram que o desempenho do *carregamento de dados* com Unicage é comparável ao do carregamento para o HDFS (o sistema de ficheiros distribuídos do Hadoop) para dados não estruturados e muito superior para dados estruturados. Para o *processamento de dados*, os resultados mostram que o desempenho do Unicage é superior ao do Spark para cargas de *procura* e inferior mas comparável ao do Spark para cargas de *agrupamento*. No entanto, as abstrações distribuídas do Hadoop permitem ao Spark executar cargas de processamento mais complicadas, como a *ordenação* e a *junção*, com resultados corretos, quando não é possível com Unicage.

Palavras-chave: Sistemas de Macrodados, Pilhas de Programas, Processamento de Dados, Avaliação Comparativa, Computação na Nuvem

Abstract

The paradigm of big data is characterized by the need to collect and process data sets of great volume, arriving at the systems with great velocity, in a variety of formats. The specialized big data processing systems, such as Apache Hadoop, Hive and Spark, offer useful abstractions to handle big data in distributed clusters, but are complex and combine a lot of libraries in deep software dependency trees. As an alternative, Unicage relies solely on the operating system and C runtime libraries to create simpler big data processing systems.

This study compares the performance of the two approaches - complex versus lean systems - with benchmarks for loading and processing of structured and unstructured data sets in a cloud computing cluster with multiple nodes hosted in the IBM Cloud. The volumes of the input data sets ranged from 64 GB to 8192 GB.

The experimental results show that the performance of *data loading* with Unicage is comparable to that of loading into HDFS (the distributed file system of Hadoop) for unstructured data and vastly superior for structured data. For *data processing*, the results show that the performance of Unicage is superior to that of Spark for *search* workloads and inferior but comparable to that of Spark for *grouping* workloads. However, the distributed abstractions of the Hadoop stack enable Spark to execute more complicated workloads, such as *sort* and *join*, with correct outputs, when it is not possible with Unicage.

Keywords: Big Data Systems, Software Stacks, Data Processing, Benchmarking, Cloud Computing

Contents

List of Tables

List of Figures

List of Listings

Glossary

AMPLab Big Data Benchmark A collection of big data micro-benchmarks for data warehousing and query processing frameworks.

Apache Hadoop A complex big data system that comprises HDFS, YARN and MapReduce.

Apache Hive A data warehousing and query processing framework for the processing of structured data over MapReduce with a SQL-based language called *HiveQL*.

Apache Kafka A distributed publish-subscribe message broker and stream processing framework.

Apache Pig A data-flow engine that simplifies MapReduce programming with an SQL-based language called *PigLatin*.

Apache Spark A complex big data system suitable for batch, query and stream processing.

Apache Storm A complex big data system suitable for stream processing.

API (Application Programming Interface) A software layer that provides applications with an interface to access the features and data of the operating system or other applications.

ASF (Apache Software Foundation) An organization that provides services and support for open-source software projects.

Batch processing Processing a bounded data set, fully available at the time of processing.

BDGS (Big Data Generation Suite) A big data generator capable of producing synthetic data sets with controllable volumes.

Benchmark A measure that serves as a point of reference for the comparison of a given performance attribute across multiple systems.

BigBench A big data end-to-end benchmark for data warehousing and query processing frameworks.

Big data Data with great volume, velocity and variety, whose collection, processing and analysis transcends the capabilities of single-machine or small cluster deployments.

BigDataBench A big data benchmark suite for batch, query and stream processing frameworks.

CLI (Command-Line Interface) A text-based user interface where users can issue commands to the operating system. The Linux *shell* establishes the interface between the user and the kernel through a CLI.

Cluster A group of two or more interconnected machines, often referred to as *nodes*.

Complex big data system A big data processing system that relies on a complex software stack characterized by abstractions like a distributed file system and a distributed resource manager.

CPU (Central Processing Unit) The component of a computer responsible for the execution of control, arithmetic and logic instructions.

Cryptographic hash function A mathematical algorithm that converts data of varying size into an array of bits with a fixed size, often referred to as a *hash* or *message digest*.

CSV (Comma Separated Values) A textual data format with a set of rows separated by newlines, each with a set of fields separated by commas.

Distributed computing Computation that takes place across a cluster of multiple nodes.

Distributed file system A system that enables storage and replication of data across multiple nodes, presenting it to the user as if stored in a single machine.

End-to-end benchmark A type of benchmark designed to evaluate a system in typical application scenarios, encompassing a complete data-flow, from initial input to final output.

Fault tolerance The quality attribute of a system that refers to how it provides availability. A fault tolerant system exploits redundancy in resources to detect and avoid system failures.

HDFS (Hadoop Distributed File System) The distributed file system of the Hadoop stack.

HiBench A big data benchmark suite for batch, query and stream processing frameworks.

Internet of Things Paradigm characterized by the large-scale deployment of multiple low-powered interconnected sensor-like devices with low hardware specifications.

JVM (Java Virtual Machine) A virtual machine that provides the runtime environment for applications compiled to the Java programming language bytecode instructions.

LeanBench A set of micro-benchmarks for both complex and lean big data systems deployed in a single-machine environment.

Lean big data system A big data processing system that relies on a lean software stack characterized by an absence of distributed abstractions, delegating most control do the developer.

MapReduce A big data processing paradigm that relies on two main operations: the *Mapper* that processes an input data set into a set of intermediate key/value pairs; and the *Reducer* that processes the intermediate key/value pairs output by the Mapper and produces output key/value pairs.

MD5 (Message Digest 5) A cryptographic hash function that generates a 128-bit message digest.

Micro-benchmark A type of benchmark designed to evaluate systems in respect to small operations.

Netdata A cluster performance observability tool.

NFS (Network File System) One of the first distributed file systems, developed by Sun Microsystems in the 1980s.

PaSh A system that increases the performance of POSIX shell scripts by automatically parallelizing computation, without code refactoring.

PDGF (Parallel Data Generation Framework) A big data generator capable of producing synthetic structured data sets with controllable volumes.

PoSh A system that increases the performance of I/O-intensive shell scripts by offloading the computation closer to the data it uses, without code refactoring.

POSIX (Portable Operating System Interface) A standard that specifies the APIs and utilities found across UNIX-like systems and other operating systems.

Query processing Processing a bounded structured data set with a query language.

RAM (Random Access Memory) The component of a computer where data in current use is stored to be quickly accessed by the CPU.

RDD (Resilient Distributed Data set) An immutable data structure used by Spark.

Scalability The quality attribute of a system that refers to how it responds to growing workloads to meet performance and cost demands.

Security The quality attribute of a system that refers to how it provides integrity and confidentiality of data.

Semi-structured data Type of data with predefined structural properties, but without a formal schema definition.

Shell A command-line interpreter that establishes the interface between the user and the kernel of the operating system.

Shell script A program comprised of commands that can be executed in UNIX operating systems. Commands can be composed in sequence, forming processing pipelines.

Software stack A set of software subsystems and their inter-dependencies.

SparkBench A big data benchmark suite designed to evaluate the performance of Spark.

SPEC (Standard Performance Evaluation Corporation) A consortium whose goal is to establish, endorse and maintain benchmarking standards for the evaluation of the performance of modern computing systems.

SQL (Structured Query Language) A standardized language for access, retrieval and manipulation of structured data in a database.

Stream processing Continuous processing of unbounded data with strict time-constraints.

Structured data Type of data with a set of predefined structural models specified with a formal schema definition.

TPC (Transaction Processing Performance Council) A consortium whose goal is to establish, endorse and maintain benchmarking standards to promote efficient and reliable transaction processing systems and databases.

Unicage A proprietary tool for shell script-based data processing solutions.

Unicage BOA (Big data Oriented Architecture) A Unicage library that enables parallel and distributed data processing with a set of specialized shell commands.

Unicage Tukubai A Unicage library for single-machine data processing with a set of specialized shell commands.

UNIX A family of multiuser, multitasking and mostly open-source operating systems that includes Linux and macOS.

Unstructured data Type of data with no predefined structural properties.

Usability The quality attribute of a system that refers to the amount of time and effort necessary to use it effectively.

VPC (Virtual Private Cloud) A service that allows the provisioning of virtual nodes in a private virtual network infrastructure for cloud computing workloads.

V properties The properties that characterize big data, including *volume*, *velocity*, *variety*, *veracity* and *value*.

XML (eXtensible Markup Language) A markup language and data format that is machine- and human-readable.

XSD (XML Schema Definition) The standard format for the specification of the schema of XML documents.

YARN (Yet Another Resource Negotiator) The distributed resource manager of the Hadoop stack.

Chapter 1

Introduction

The amount of digital information available is growing exponentially, as more data is captured and retained in information systems. This change has caused a data processing paradigmn shift. The *big data* paradigm will be further magnified by the rise of the Internet of Things and the last generations of 5G cellular technology, with a vast amount of small, connected devices producing large amounts of information. Juniper Research estimates that in 2024, the number of Internet of Things connections will reach the 83×10^9 mark, an increase of 137% from the 35×10^9 connections observed in 2020 [1]. Statista estimates that in 2024, 147 zettabytes[1] of data will be created, an increase of 129% from the estimated 64.2 zettabytes created in 2020 [2]. Big data systems and processing tools are at the center of this rising trend [3].

Big data is structured, semi-structured and unstructured data, created in high volume, high velocity and in a wide variety of formats and sources, whose collection, analysis and transformation into value transcends the capabilities of single-machine or small cluster architectures [4]. Pappas et al. [5] emphasize how companies are growing aware of the business value of data and of the competitive edge they can gain from it. Beyond retaining big data, its value is heightened by the extraction of knowledge that is only possible with big data systems. These systems often offer mechanisms to meet various quality attributes, such as:

- *Performance*: how the system meets the required response time and throughput;
- *Scalability*: how the system responds to performance and cost demands;
- *Fault tolerance*: how the system provides availability;
- *Security*: how the system provides integrity and confidentiality of data;
- *Usability*: how much time and effort is necessary to use the system effectively.

[1] 1 zettabyte = 1×10^{12} gigabytes = 1×10^{21} bytes

We classify big data systems based on the complexity of their software stacks, which is characterized by the number of software subsystems. A software subsystem is a cohesive and re-usable unit with a clear purpose. Generally, a higher number of subsystems results in more inter-dependency, also referred to as *coupling*, and consequently increased complexity [6].

Complex big data systems rely on complex software stacks to provide data processing services on top of clusters of machines, and address the aforementioned quality attributes with high-level abstractions that allow ease of use through simple programming models. Apache Hadoop, Hive and Spark are examples of complex big data systems. HDFS and YARN are subsystems prominently featured in their software stacks.

Lean big data systems leverage the capabilities of the operating system and hardware directly, and feature lean software stacks. Unicage is a set of data management commands that extends the set of available commands in Linux and can be used to build lean big data processing applications. The terms *lean implementation* and *lean approach* are also used throughout this document to refer to lean big data systems.

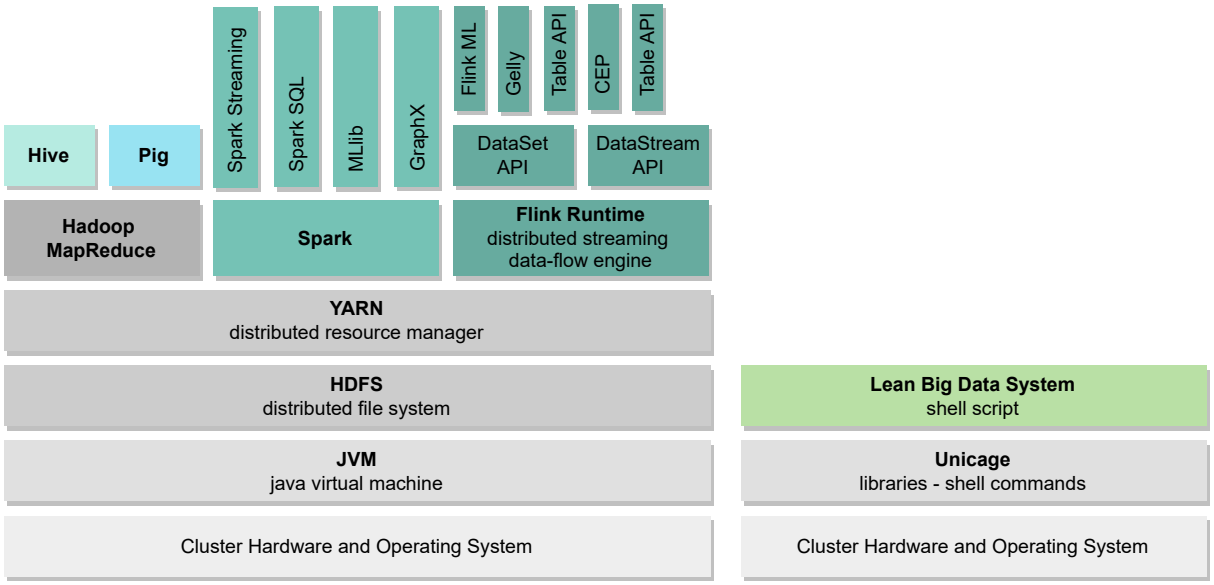


Figure 1.1: Software stacks of complex big data systems provided by the Apache Software Foundation (left); and of a Unicage lean big data system (right).

Figure 1.1 shows the software stacks of some widely used complex big data systems, including Hadoop MapReduce, Hive, Pig, Spark and Flink; and of a lean big data system implemented with Unicage. The dichotomy in software stack complexity is highlighted by the considerable difference in the number of subsystems and their inter-dependencies.

These systems are not without their trade-offs. Trivedi et al. [7] showed that the software stacking employed by complex big data systems like Spark, which shares many components

with the Hadoop stack, leads to bottlenecks that hinder clusters from taking full advantage of fast networking speeds, as the software cannot provide the required processing throughput. Furthermore, complex big data systems are not suitable for deployment in low-cost devices with low hardware specifications. Meanwhile, the simplicity of the lean software stacks employed by lean big data systems and the absence of abstractions imply the delegation of most of the control to the developer, who incidentally has to implement whichever mechanisms are needed to ensure the required quality attributes for the tackled use-case are met, resulting in tailored solutions.

There is a lack of published studies comparing complex and lean big data systems in all fronts, including performance, fault tolerance and security. LeanBench [8] is, to our knowledge, the only purposeful study, and has shown that the performance of lean implementations with Unicage can be superior to the performance of Hadoop and Hive. However, this study was limited to single-machine configurations, which prevented experiments with larger data sets. A comparative basis between complex and lean big data systems in a distributed environment will be an asset to further develop studies assessing the benefits of the two approaches.

1.1 Contributions

The primary contribution of this study is a refined performance analysis and comparison between complex big data systems like Apache Hadoop, Hive and Spark, and lean big data systems built with Unicage, in a cloud computing cluster environment. The study assesses the impacts that distributed system abstractions have in various loading and processing workloads applied to data sets with volumes ranging from 64 GB to 8192 GB. This was achieved with a big data benchmark suite, with a clear definition of conceptual models for the data-flow [9] and the re-use of existing benchmarks, such as LeanBench [8] and BigDataBench [10, 11].

1.2 Thesis Outline

This document is organized in chapters. This chapter introduced the context and main motivations of the study. Chapter 2 contains an extensive background on all technologies mentioned and used throughout the study. Chapter 3 addresses previous works that inspired the study. Chapter 4 describes the implementations, environments and conducted experiments. Chapter 5 provides the report and discussion of the collected results. Chapter 6 presents the main conclusions and achievements of the study, along with a retrospective reflection, and proposes directions for future work.

Chapter 2

Background

This chapter provides a detailed explanation of all the technology mentioned and used throughout this study. An overview of the important theoretical concepts is presented in Section 2.1. Complex big data systems are addressed in Section 2.2. Unicage and why it is a good representative of lean big data systems is addressed in Section 2.3. A summary of the chapter is provided in Section 2.4.

2.1 Theoretical Overview

Big data processing environments are characterized by various factors: the properties of big data, described in Section 2.1.1; the types of data, described in Section 2.1.2; and the types of processing workloads, described in Section 2.1.3. Big data systems often rely on distributed file systems to store data sets efficiently across multiple nodes, therefore distributed file systems are discussed in Section 2.1.4. Finally, Section 2.1.5 introduces the basic building blocks of shell scripting, as it is the foundation of the development of Unicage systems.

2.1.1 Big Data Properties

Big data is characterized by a set of properties often addressed as the *V properties*. These highlight the difference between big data and common data, and are defined as follows:

- *Volume*: refers to the size of data, for instance, when saying that a data set has a volume of 64 GB;
- *Velocity*: refers to the input rate of data, for instance, when saying that a data set arrives at a system at a rate of 1 GB/s;
- *Variety*: refers to the spectrum of possible formats and sources of data;

- *Veracity*: refers to the accuracy, truthfulness and trustworthiness of data, its sources and the processing workloads;
- *Value*: refers to the usefulness of the data and can be specified in monetary value.

2.1.2 Data Set Types

In the big data paradigm, data sets come in a wide variety of formats and can be classified as *structured*, *semi-structured* and *unstructured* data.

Structured data has a set of predefined structural models specified with a formal schema definition. This data can be stored in tabular format in databases. Formats like *eXtensible Markup Language* (XML) and *Comma Separated Values* (CSV) are considered structured formats when a formal schema definition is present. For example, the schema of XML documents is specified with *XML Schema Definition* (XSD).

Semi-structured data has predefined structural properties, such as delimiters (e.g., tags in XML), but has no formal schema definition.

Unstructured data has no predefined data model. For example, video, audio, image and text documents are unstructured data formats, even though they have some internal structure required to render their contents.

2.1.3 Workload Types

Big data systems target one or more types of data processing workloads, including *batch processing*, *query processing* and *stream processing*. The intensity of a data processing workload is characterized by the amount of work performed by the big data systems to fulfill it.

A batch processing workload consists in offline processing of a bounded data set of limited size, fully available at the time of processing. The intensity of these workloads is mostly influenced by the volume of the input data sets.

A query processing workload consists in using a query language, such as *Structured Query Language* (SQL), to perform batch processing of structured data normalized as tables in a database. Like batch processing, the intensity of query processing workloads is mostly influenced by the volume of the input data sets.

A stream processing workload consist in online processing of an unbounded data set that is not fully available at the time of processing, but is incoming in a continuous flow of data. The processing workload itself is also a continuous process, subject to very strict time constraints. The intensity of these workloads is mostly influenced by the velocity of the input data sets.

There are two distinct types of processing granularity in stream processing:

- *Micro-batch*: each record results from the partitioning of the stream in small batches;
- *Event-based*: each record corresponds to each arriving event individually.

Stream processing can be further classified based on the relations between streaming records. In *stateless streaming*, the state of each record is independent from previous records, while in *stateful streaming*, the state of each record depends on the state of previous records.

2.1.4 Distributed File Systems

Distributed file systems are a requirement for the retention of data in big data systems. There are two concepts associated with the structure of a file that are central to the study of distributed file systems:

- *Raw data*: the raw content of the file, i.e., the useful information for the user;
- *Metadata*: additional information (e.g., the *i-node* of a file) that eases file location and use within the file system, i.e., useful information for the machine operation.

A distributed file system allows users to store files remotely across a cluster of machines, while still presenting the files to the user as if they were local files in a single machine. Distributed file systems have been around for nearly four decades. Network File System (NFS) was a pioneering file system and communication protocol developed by Sun Microsystems in the 1980s [12], and is still used today. Google File System (GFS) was the original proprietary distributed file system developed by Google to meet the sudden demands from the turn-of-the-century Internet data explosion [13], and has since been replaced with the also proprietary Colossus [14]. In the mid 2000s, big data and cloud computing motivated the emergence of open-source and cloud-suitable distributed file systems.

Distributed file systems mostly differ in the way data and metadata are persisted and managed across the hosting cluster. HDFS [15], which stands for Hadoop Distributed File System, is of special relevance, as it is the distributed file system of Hadoop and was extensively used throughout this study. It is addressed in detail in Section 2.2.1, along with the other subsystems of the Hadoop software stack.

2.1.5 UNIX: Shell Scripting and Command Pipelining

UNIX is a family of operating systems originated in the 1970s following a common philosophy [16]. Standard UNIX-like systems comply with the POSIX[1] standard, as defined by the Institute of Electrical and Electronics Engineers (IEEE) in the 1980s. The *shell* is one of the most powerful tools of UNIX and establishes the interface between the users and the kernel[2]. When a user issues a shell command through the *command-line interface* (CLI), it is interpreted by the shell into a set of system calls to be processed by the kernel. Linux is a modern and popular UNIX-like operating system and features the `bash` shell (acronym for "Bourne Again SHell"). In UNIX, everything is perceivable as files, which enables users to have control over their environment and programs. In particular, shell commands are also files, and users can create their own commands or chains of commands through pipelining and shell scripting. Inspired by Shotts [17], we highlight the building blocks of shell scripting, as the concept is central to the development using Unicage.

Shell Commands

Shell commands are user-defined programs that can be invoked directly from the command-line. A command sends its results to a special file called *stdout* (or *standard output*) and its status messages to another special file called *stderr* (or *standard error*). Both the output and exit statuses can be redirected to other files. Additionally, shell commands always emit an exit status code to a global variable called `$?`. The exit status of a command can be 0 for success, or an integer from 1 to 255 for errors.

Command Pipelines

Command pipelines allow commands to be linked together in a data-flow, where the output of a first command is input to a second command, and so on. A command pipeline is expressed semantically with the | operator (called a *pipe*). Each command in a pipeline is run in a child process of the shell the pipeline was started on. If possible, the commands in a pipeline are executed in parallel. The exit status of a pipeline is an array with the exit statuses of each command, ordered from first to last, and is stored in the variable `${PIPESTATUS[@]}`. A pipeline is considered to execute without errors if the sum of the values in this array is equal

[1] **POSIX**: a standard, formally known as IEEE 1003, that specifies the APIs and utilities found across UNIX-like systems and other operating systems.

[2] **kernel**: the core component of an operating system, responsible for system call control, and process, memory and device management.

to 0. Figure 2.1 shows the data-flow of a basic command pipeline that displays the five largest files in a given directory.

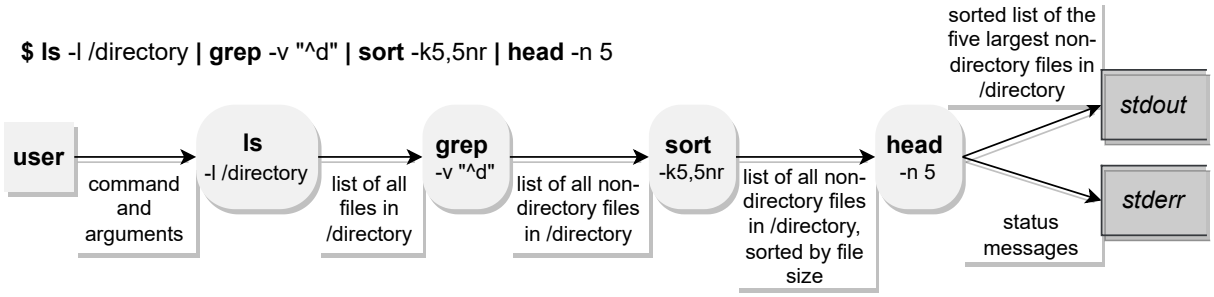


Figure 2.1: Data-flow diagram of an example command pipeline in UNIX.

Shell Scripts

Shell scripting is a programming language that allows the combination of shell commands and pipelines to create modular programs called *shell scripts*. With execution permissions, a shell script can be executed with `./script-name.sh <argument1> <argument2> ... <argumentN>`, and each argument is stored, within the shell script, in the positional parameters `$1`, `$2`, ..., `$N`, respectively. The first line of a shell script is informally called a *shebang*, and it tells the operating system which interpreter to use - if the script is to be run in `bash`, the first line must be `#!/bin/bash`. Listing 2.1 materializes the pipeline in Figure 2.1 in a small shell script and, in the end, echoes to *stdout* the statuses of the commands in the pipeline. Listing 2.2 shows a possible output of a successful execution of the script in the shell CLI, where the fifth and ninth columns specify the sizes and relative path names of the files, respectively.

```
1 #!/bin/bash
2
3 ls -l $1         | # fetch list of all files in $1
4 grep -v "^d"     | # filter out directories
5 sort -k5,5nr     | # sort by file size
6 head -n 5          # trim list down to first 5 rows
7
8 echo ${PIPESTATUS[@]} # echo command statuses to stdout
```

Listing 2.1: Shell script: listing the five largest files in a directory.

```
$ ./script.sh /directory
-rw-rw-r-- 1 user user 472 Set  8 22:40 fileS
-rw-rw-r-- 1 user user 290 Set  8 22:40 fileO
-rw-rw-r-- 1 user user 240 Set  8 22:40 fileR
-rw-rw-r-- 1 user user 220 Set  8 22:40 fileT
-rw-rw-r-- 1 user user 211 Set  8 22:40 fileD
0 0 0 0
```

Listing 2.2: Shell CLI: Successful execution of the shell script in Listing 2.1.

Shell scripting was not only central to the development with Unicage, but also to various system administration activities, including the deployment and configuration of some of the systems addressed in Section 2.2 and Section 2.3.

Parallel and Distributed Shell Scripting - PASH and POSH

Recent systems like PASH [18] and POSH [19] have shown promising results in the performance optimization of shell scripts through the parallelization and cluster distribution of computation, without requiring code refactoring [20].

PASH allows the automated parallelization of POSIX shell scripts. It performs transformations based on the parallelizability properties of the commands in the shell script, preserving its data-flow and semantics. The performance of shell scripts parallelized with PASH was shown to be an average of 6.7 times superior to that of the unmodified scripts.

POSH allows the optimization of I/O-intensive shell scripts, i.e., shell scripts that incur in considerable transfers of data over the network. Its principle is to offload the computation closer to the data it operates on, minimizing data movement. Individual shell commands are offloaded based on an annotation language that discerns the files each command will access during its runtime. POSH was tested against shell scripts with access to remote storage through NFS. The performance of the shell scripts offloaded with POSH was shown to be 1.6 to 15 times superior.

2.2 Complex Big Data Systems

The most widely used systems for big data processing in cloud environments feature complex software stacks. These systems have high-level data and computational abstractions to ease the development process and to provide quality attributes, such as scalability, fault tolerance,

security and usability. We call these *complex big data systems* due to the size and complexity of their software stacks, which feature multiple highly coupled subsystems.

The Apache Software Foundation (ASF)[3] is a public organization established in 1999 with the purpose of providing support and community for collaborative open-source software projects. Among the many projects hosted by the ASF are a variety of complex big data systems, such as Apache Hadoop, Hive and Spark. This section primarily addresses these complex big data systems and their subsystems.

2.2.1 Apache Hadoop

Apache Hadoop[4] is a system designed for big data processing in cluster environments using the *MapReduce* programming model [21, 22]. It features a complex software stack, offering storage and distributed computing abstractions for complex big data processing tasks. A typical Hadoop cluster features Hadoop Distributed File System (HDFS) [15] as the underlying distributed file system, YARN (acronym for "Yet Another Resource Negotiator") [23] as the distributed resource manager, and the whole stack runs on top of the Java Virtual Machine (JVM). These subsystems are shared by Apache Hive, Spark, and other systems built on top of Hadoop.

HDFS

Purposefully developed as a component of Hadoop, HDFS is an open-source implementation of a distributed file system inspired by GFS, and its interface presents the user with the abstraction of a single global namespace. The typical HDFS cluster has a leader/follower architecture with, at least, one of each of the following nodes:

- *NameNodes*: the dedicated leader nodes where the metadata of files is stored;
- *DataNodes*: the worker nodes where the raw data of files is stored and replicated with a default replication factor of 3, i.e., each data block has, at least, three copies.

In HDFS, large files are partitioned in blocks with a default size of 128 MB and replicated across the available DataNodes according to a storage load balancing algorithm.

[3] https://www.apache.org/foundation/
[4] https://hadoop.apache.org/

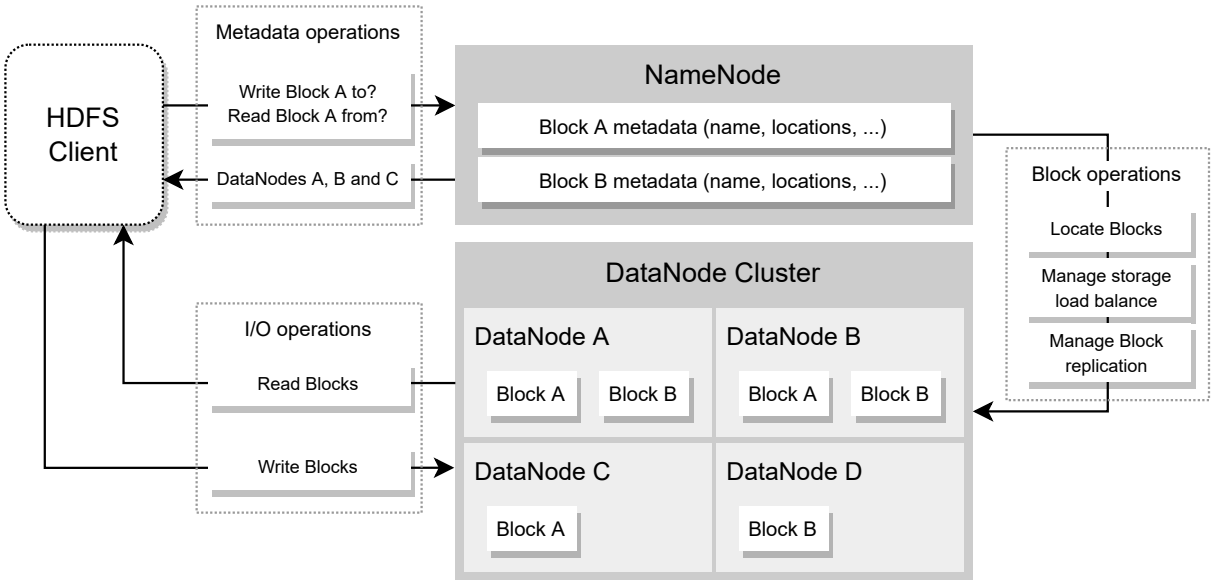


Figure 2.2: Architecture of HDFS.

Figure 2.2 illustrates the architecture of a typical HDFS cluster, the interactions between its components and how file blocks are replicated in the DataNodes. To read a file block, the HDFS Client asks the NameNode for the DataNodes where that block is located, and retrieves it from one of those nodes. To write a file block, the HDFS Client asks the NameNode for the list of DataNodes it should write the file block to, and writes it to all of those nodes. The `get` and `put` commands can be issued from an HDFS Client to trigger the read and write operations, respectively.

HDFS is inefficient when it comes to the read and write performance of large quantities of files whose volumes are largely inferior to the size of each file block [24]. This scenario creates large numbers of file blocks, which increases the latencies associated with metadata management and file block lookup. Therefore, HDFS has better performance when the data sets are or can be converted into files large enough to fill entire blocks in the file system.

YARN

Hadoop, as an open-source implementation of MapReduce, was frequently subject to custom extensions made by developers to manage the physical resources of the clusters. YARN is the distributed resource manager of the Hadoop system and was implemented to avoid the need for these ad-hoc extensions by decoupling resource management from the MapReduce programming framework. YARN provides load balancing of resource utilization and cluster scalability for Hadoop applications, as well as *multi-tenancy*, allowing multiple logically separated

applications to execute in a shared Hadoop environment, using the same physical resources.

In YARN, resources like memory, CPU and disk space are bundled in *containers*, where YARN applications are allowed to run. The balanced utilization of system resources is assured by the following three components:

- *ResourceManager*: a per-cluster service responsible for the resource allocation throughout the corresponding Hadoop cluster;
- *NodeManager*: a per-node service responsible for updating the ResourceManager with the status of the corresponding Hadoop node, where it executes containers according to the resources available in the node itself;
- *ApplicationMaster*: a per-application library that negociates resources with the ResourceManager and works with the NodeManager to execute, manage and monitor the YARN application containers.

MapReduce

MapReduce is a programming model designed for batch processing of large volumes of data. A MapReduce workload is started as a YARN application. It takes a data set as input and produces key/value pairs as output. The typical MapReduce application has two operations that must be specified by the user:

- *Mapper*: an operation that processes the input data set and produces intermediate key/value pairs;
- *Reducer*: an operation that processes the intermediate key/value pairs output by the Mapper operation and produces output key/value pairs.

MapReduce performs two other operations that, depending on the format of the input data set and available system resources, may also need to be specified by the user:

- *Split*: an operation that splits the input data set in chunks of a user-defined size, to be processed by individual Mappers;
- *Shuffle*: an operation that groups the intermediate key/value pairs output by the Mapper operation in logic key/value pairs, sorted by key, to be processed by individual Reducers.

The splitting of the input data set and shuffling of the intermediate key/value pairs allows Mapper and Reducer operations to be parallelized across the Hadoop cluster.

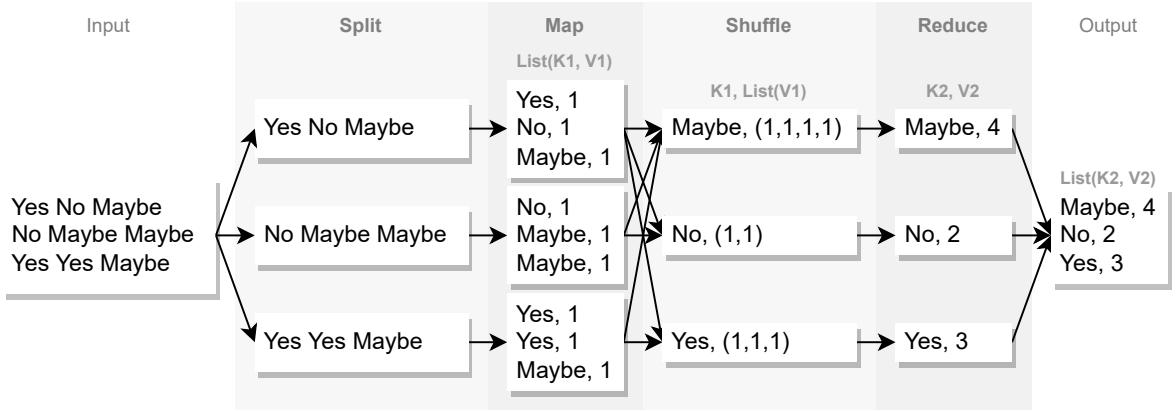


Figure 2.3: Diagram of a wordcount example in MapReduce.

Figure 2.3 highlights all of the aforementioned operations in a simple wordcount program in MapReduce, where the input data set contains a set of three unique words: ”Yes”, ”No” and ”Maybe”. This data set is split in three chunks, to be processed by three individual Mappers that can be parallelized. The Mappers match each occurrence of a word with the value 1. The outputs of the Mappers are shuffled in three groups, sorted by key, to be processed by three individual Reducers that can also be parallelized. In this case, the groups are logically separated by word, and each group has a single key/value pair, so they are sorted by default. The Reducers aggregate the counts of each word and output the final list of key/value pairs.

2.2.2 Apache Hive

Apache Hive[5] is a data warehousing system open-sourced by Facebook in 2008 and officially released in 2010 [25, 26]. It is built on top of Hadoop and enables query processing with large structured data sets through a declarative query language inspired by SQL called *HiveQL*. HiveQL queries are parsed, optimized and compiled into MapReduce jobs to be processed by Hadoop. A structured data set stored in HDFS can be represented as *tables* in the backend database of Hive, which is, by default, Apache Derby[6]. This does not require copying the data set and allows Hive to leverage the fault tolerance and load balancing capabilities of HDFS. There are three data structures in Hive, with distinct purposes [27]:

- *Tables*: reminiscent of tables in relational databases, divided in structured *columns*;

[5]https://hive.apache.org/
[6]https://db.apache.org/derby/index.html

- *Partitions*: result from partitioning large tables in smaller tables, based on the values of one or more columns;
- *Buckets*: result from partitioning large tables or partitions in small files, based on the value of a single column.

Contrarily to partitioning, bucketing can only be performed on one of the columns of a table, and the number of buckets can be specified by the user. Listing 2.3 is an example of a statement in HiveQL, featuring table creation, partitioning and bucketing.

```
1 -- Creating table named 'person':
2 create table person (name string, age int)
3 -- Partitioning based in column named 'country':
4 partitioned by (country string)
5 -- Bucketing in 10 buckets, based in column named 'age':
6 clustered by age into 10 buckets;
```

Listing 2.3: HiveQL statement: table creation, partitioning and bucketing

Every statement in Hive is executed in the context of a *session*. A session is initialized either when the user enters the interactive command-line interface of Hive, or when a statement is submitted to Hive from the shell. The smooth integration of Hive with Hadoop MapReduce is dependent in an architecture with four key components:

- *Driver*: responsible for handling and monitoring sessions, and for managing the lifecycle of HiveQL statements;
- *Compiler*: responsible for compiling HiveQL statements into *direct acyclic graphs*[7] of operations to be executed as MapReduce jobs by Hadoop;
- *Metastore*: responsible for storing metadata information about the tables, partitions and buckets;
- *Execution Engine*: establishes the interface between Hive and Hadoop, and is responsible for executing the compiled jobs.

Figure 2.4 illustrates the architecture of Hive and the interactions between its components. A Hive Client submits a HiveQL statement to the Driver. The Driver requests the execution plan to the Compiler, who compiles the statement into a direct acyclic graph of MapReduce

[7] **direct acyclic graph**: a directed graph characterized by a set of vertices, unidirectional edges and the absence of cycles.

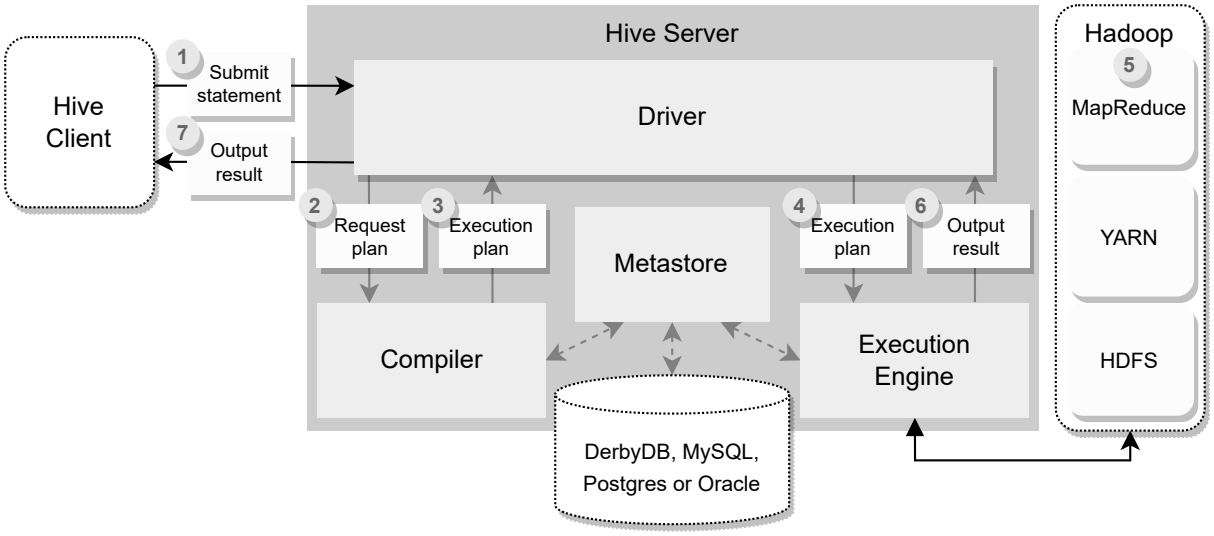


Figure 2.4: Architecture of Apache Hive.

operations. The Driver forwards this execution plan to the Execution Engine, which communicates with the Hadoop stack to coordinate the execution of the MapReduce jobs. The output is retrieved and returned to the Hive Client.

2.2.3 Apache Spark

Apache Spark[8] is a system introduced in 2010 [28] and is designed for iterative and interactive big data analysis. The motivation of Spark stems from the inefficiency of Hadoop for iterative algorithms that read data repeatedly, as each MapReduce job reloads data from disk, which is slower than memory access [29]. Spark addresses these shortcomings by compiling jobs into direct acyclic graphs of operations to be performed on specialized immutable data structures called *resilient distributed data sets* (RDDs) [30]. RDDs are partitioned across the nodes in a Spark cluster and can be rebuilt if nodes fail or partitions are lost. RDD partitions are stored in memory and loaded on demand, which provides a significant increase in performance when compared with the MapReduce approach. When Spark runs out of memory to cache RDD partitions, the default behavior is to use the lineage information of these partitions to recompute them on demand. An alternative is to configure Spark to store the partitions that do not fit in memory on disk, and to read them when needed. The trade-off is that both of these approaches are known to cause performance degradation.

As a result of the extension and optimization of RDDs, Spark also features other data structures, such as *DataFrames*, which are immutable distributed collections of tables with named

[8] https://spark.apache.org/

columns that enable query processing; and *DataSets*, which are collections of typed Java objects that enable query optimization and compile-time type safety[9].

Spark can be integrated with the Hadoop stack to use HDFS and YARN. Every Spark application runs in the context of a *SparkContext*. A typical Spark cluster features the following processes:

- *Driver*: the leader process that is responsible for creating, executing and coordinating the SparkContext, compiling jobs into direct acyclic graphs, planning tasks, and communicating with YARN to allocate resources in the Executor nodes;
- *Executors*: the worker processes that are responsible for the execution of the tasks planned by the Driver, and for saving the outputs in HDFS.

A Spark application can be deployed in either *client* or *cluster* mode. Client mode is mostly used for debugging, as the Driver process executes in the machine the Spark application is submitted from. Cluster mode is used in production environments, as the Driver process executes in one of the worker nodes.

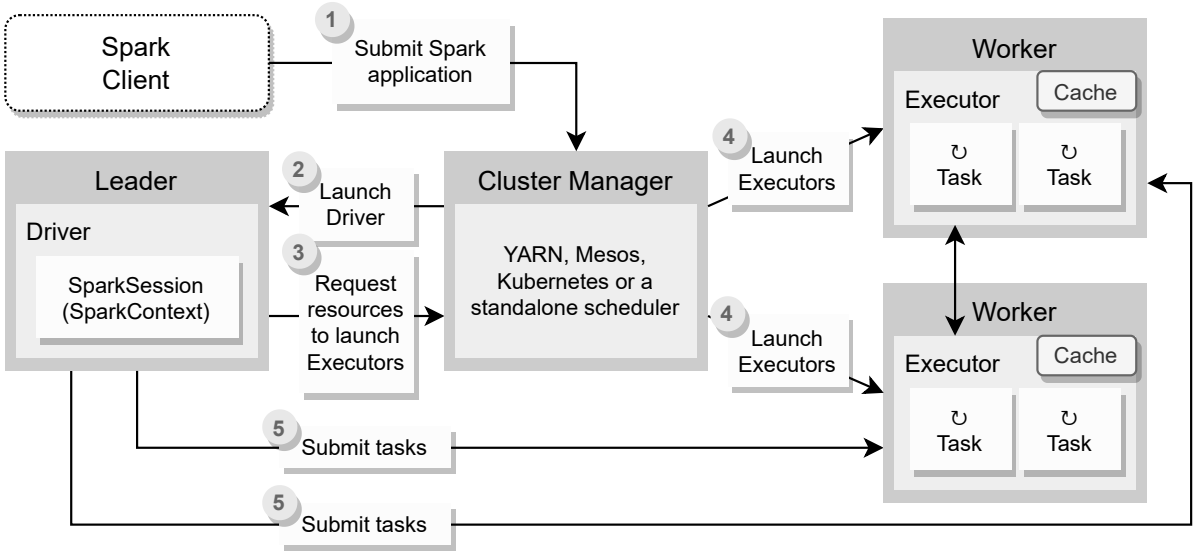


Figure 2.5: Architecture of Apache Spark.

Figure 2.5 illustrates the generalized architecture of a Spark cluster in the context of a running application. Spark can be used with a variety of distributed resource managers, but let us assume it is using YARN in cluster mode. As such, the leader node executing the Driver is one of the DataNodes; the cluster manager is the NameNode executing the ResourceManager

[9]**compile-time type safety**: errors related with the usage of invalid types are raised at compile time, and not at runtime.

of YARN; the workers are also DataNodes; and much like MapReduce jobs, Spark jobs are submitted as YARN applications. The Driver executes within the ApplicationMaster of YARN and is started once a Spark job is submitted to the cluster. The ApplicationMaster negotiates resources with the ResourceManager and works together with the NodeManager of each DataNode to start the Executors, instantiated as YARN application containers. With this initial setup, the Driver creates the direct acyclic graph, schedules its stages, and submits the tasks to the Executor nodes.

The architecture of Spark, with the immutable and fault tolerant nature of its data structures, allowed it to expand beyond batch processing to feature various domain-specific libraries:

- *Spark SQL* [31]: framework for query processing, through the use of DataFrames;
- *Spark Streaming* [32]: framework for micro-batch stream processing through the use of specialized data structures called *discretized streams* (D-streams);
- *Spark Structured Streaming* [33]: framework that reworks stream processing in Spark through the use of DataFrames and DataSets tailored for streaming. Each row in a DataFrame or DataSet is processed as an individual streaming record, making stream processing in Spark closer to event-based stream processing;
- *GraphX* [34]: framework for the processing of graph data;
- *MLlib* [35]: framework for machine learning.

2.2.4 Apache Kafka

Apache Kafka[10] is a distributed message broker open-sourced by LinkedIn in 2011. Kafka is used as a *data ingestion tool*[11] for stream processing workloads implemented in Spark and other big data systems, but it also has its own scalable and fault tolerant event-based stream processing library called *Kafta Streams*. Kafka does not rely in the Hadoop stack directly, but is supported by its own complex software stack running in the JVM. The typical Kafka architecture has four key components:

- *Topics*: logical stream categories to which messages can be *pushed* and from which messages can be *pulled*;

[10] https://kafka.apache.org/
[11] **data ingestion tool**: a tool that extracts streaming data and transfers it into a storage or logic layer.

- *Producers*: nodes that push messages to the Kafka cluster, i.e., publish messages to a given Topic;

- *Brokers*: nodes that make up the Kafka cluster and are responsible for the storage and replication of Topics;

- *Consumers*: nodes that subscribe to Topics and pull the corresponding messages from the Kafka cluster.

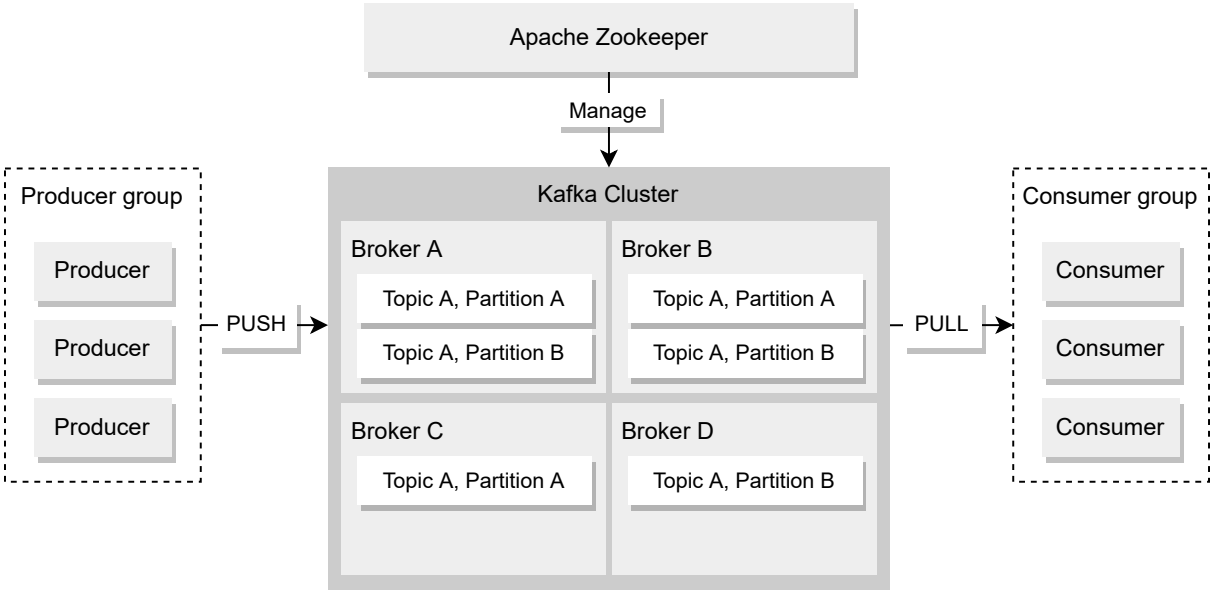


Figure 2.6: Architecture of Apache Kafka.

Figure 2.6 illustrates the architecture of a typical Kafka cluster, the interactions between its components, and how Topics are replicated in the Broker nodes. Topics are split in partitions, much like files in HDFS are split in file blocks. Partitions are append-only immutable logs, preserving the order at which streaming messages arrive. Each partition is maintained by one Broker, assigned as its leader to coordinate its replication; and by other Brokers, assigned as followers, where the partition is replicated. In Figure 2.6, partitions are replicated thrice, meaning each partition has a leader Broker and two follower Brokers (e.g., Broker A may be the leader for partition A of Topic A, and Brokers B and C may be its followers). Apache Zookeeper [36] is used to monitor the statuses of the Brokers. A Producer writes in a Topic partition by pushing its messages to the corresponding leader Broker, which then updates the follower Brokers. A Consumer reads from a Topic partition by pulling the messages from the corresponding leader Broker.

2.2.5 Other Complex Big Data Systems

Some big data systems that were not directly featured in this study are briefly described here.

Apache Pig

Apache Pig[12] is a data-flow engine open-sourced by Yahoo in 2007 and officially released in 2008 [37]. It offers a high-level abstraction through *PigLatin* [38], an SQL-based procedural data-flow language that allows MapReduce jobs to be programmed with considerably less effort. Despite also being a query language, PigLatin supports batch processing of semi-structured and unstructured data. The *Pig Engine* is the component responsible for compiling PigLatin statements into direct acyclic graphs of MapReduce jobs.

Apache Storm

Apache Storm[13] is a distributed stream processing framework running on top of the JVM, and can be integrated with HDFS and YARN. A Storm cluster is similar to a Hadoop cluster and is comprised of a leader node called *Nimbus*, which relies on Apache ZooKeeper to monitor worker notes called *Supervisors*. Storm applications are represented as direct acyclic graphs of operations called *topologies*. A Storm topology is characterized by two types of graph vertices: *Spouts* are sources of data streams, and *Bolts* process data streams originated from Spouts or output by other Bolts. Clients submit topologies to the Nimbus node, which is responsible for coordinating their execution in the Supervisor nodes.

Apache Flink

Apache Flink[14] [39] is both a batch and stream processing framework, and unifies both types of workload in a single model. The motivation of Flink stems from the perspective that a bounded data set is a special case of an unbounded data set, and by extension, batch and stream processing can be handled the same way, as both types of workloads are represented as streaming data-flows. In Flink, the *DataSet API* is used for batch processing, and the *DataStream API* is used for stream processing. Additionally, Flink features its own domain-specific libraries, with the *Table APIs* for query processing; *FlinkCEP* for complex event processing[15]; *Gelly* for the processing of graph data; and *FlinkML* for machine learning.

[12] https://pig.apache.org/
[13] https://storm.apache.org/
[14] https://flink.apache.org/
[15] **complex event processing**: a set of proposed techniques to extract desired information from event-based streams.

2.2.6 Comparing the Complex Big Data Systems

The complex big data systems that integrate with Hadoop feature shared core mechanisms to provide scalability, fault tolerance, security and usability. Kafka, as a standalone system, provides its own mechanisms. All of these systems can be compared from the perspective of the types of workload they are tailored for. Table 2.1 presents a comparison between all surveyed complex big data systems, sorted by the open-sourcing year.

		Workload types			
Big data system	Software stack integrations	Batch processing	Query processing	Stream processing	Open-sourcing year
Hadoop MapReduce	HDFS and YARN	Yes	No	No	2006
Pig	Hadoop MapReduce	Yes	Yes	No	2007
Hive	Hadoop MapReduce	No	Yes	No	2008
Spark	HDFS and YARN	Yes	Yes	Yes	2010
Kafka	Zookeeper	No	No	Yes	2011
Flink	HDFS and YARN	Yes	Yes	Yes	2011
Storm	Zookeeper, HDFS and YARN	No	No	Yes	2012

Table 2.1: Comparison between complex big data systems.

These systems exist as the result of a traceable evolution of necessity. Formerly, there was little need beyond batch processing, and MapReduce was the leading paradigm. To ease the use of MapReduce for big data processing, systems like HDFS and YARN were proposed to provide distributed storage and resource usage management spanning multiple nodes. MapReduce was infamously complex to program, so systems like Hive and Pig were proposed to hide some of this programming complexity and offered better solutions for structured data. As the volumes of big data increased and the MapReduce paradigm became obsolete, systems like Spark proposed more efficient alternatives for batch processing. As the focus shifted from batch processing to stream processing, systems like Storm and Flink were proposed, and existing systems like Spark started to expand with stream processing libraries. The velocity at which streaming data arrives at the systems introduced challenges associated with its retention, so data ingestion tools like Kafka appeared to facilitate the efficient exchange of streaming data, while preserving its veracity.

In the recent years, Spark and Flink have been some of the most popular projects of the ASF [40, 41].

2.3 Lean Big Data Systems

Lean big data systems feature lean software stacks, but offer less abstractions than complex big data systems. The absence of abstractions implies an extra investment from the developer to implement security, fault tolerance, parallelism, and other mechanisms, if needed.

Unicage[16] is a good representative of the concept of a lean big data system despite being a proprietary tool, as its approach can be followed by other open-source alternatives. It embraces the UNIX fundamentals and offers a set of proprietary commands written in the C language. These commands can be used alongside Linux shell commands to implement shell script-based data processing pipelines for text file-based data sets. The C language uses less memory and consumes less energy than other languages [42], and therefore saves evergy in the servers and is suitable for the deployment in low-powered devices with low hardware specifications. Unicage features a base library for single-machine data processing called *Tukubai*, and an extension of this library with additional commands for parallel and distributed data processing called *Big data Oriented Architecture* (BOA).

2.3.1 Unicage Tukubai

Unicage Tukubai features over 200 shell commands tailored for data processing in a single machine. Some of these commands are listed in Table 2.2, along with their purposes.

Command	Description
`msort`	Performs in-memory sort of records using the merge-sort[17] algorithm.
`lcnt`	Counts the number of records in a file.
`dmerge`	Merges two sorted files based on specified columns.
`self`	Selects and reorders the columns in a file.
`sm2`	Aggregates records grouped by specified columns.

Table 2.2: Examples of Unicage Tukubai commands.

2.3.2 Unicage BOA

Unicage BOA features shell commands analogous to a subset of commands in Tukubai, but tailored for parallel and distributed big data processing in cluster environments.

[16] https://unicage.eu/

[17] **merge-sort**: a recursive sorting algorithm that applies the principle of *divide-and-conquer* [43].

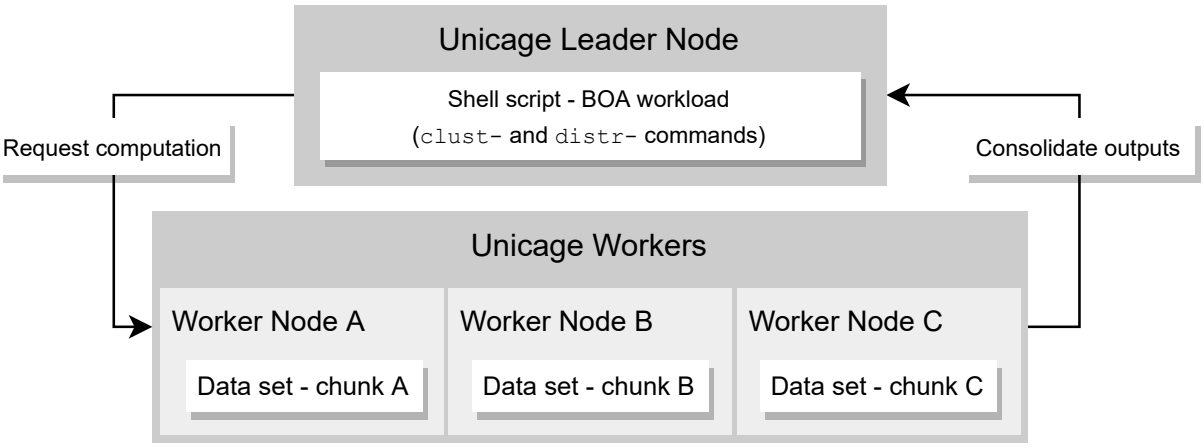


Figure 2.7: Architecture of Unicage BOA.

A typical BOA cluster has a leader/follower architecture, as represented in Figure 2.7. In a BOA cluster, data sets are equally partitioned across the worker nodes, and BOA commands are issued from the leader node. Each worker is in charge of processing its chunk of the data set. The output of each worker is sent back to the leader node, where it is consolidated, depending on the workload.

All nodes in a BOA cluster must be interconnected. Their IP addresses or hostnames are passed as arguments to BOA commands to specify the nodes that should partake in the computation. There are three types of BOA commands:

- `para-` *commands*: aimed at parallel data processing in a single machine with a multi-core processor;
- `clust-` *commands*: aimed at parallel and distributed data processing in a cluster environment;
- `distr-` *commands*: similar to `clust-` commands, but the leader node does not partake in the computation.

Table 2.3 highlights some BOA commands that are analogous to the Tukubai commands described in Table 2.2. The `lcnt` Tukubai command does not have BOA equivalents.

BOA also features a set of `distr-` commands implementing file system operations, such as `mkdir`, `rm` and `cp`, providing an interface similar to that of a distributed file system. The `distr-distr` command is noteworthy, as it is used to distribute data across the BOA cluster.

	BOA commands		
Tukubai command	**para-**	**clust-**	**distr-**
msort	n/a	clust-msort	distr-msort
dmerge	n/a	n/a	distr-dmerge
self	para-self	clust-self	distr-self
sm2	para-sm2	clust-sm2	distr-sm2

Table 2.3: Examples of Unicage BOA commands.

2.4 Chapter Summary

This chapter introduced relevant concepts for the understanding of this study, starting with the properties that characterize data, i.e., volume, velocity, variety, veracity and value; the structural categorization of data, i.e., structured, semi-structured and unstructured data; and the types of data processing, i.e., batch, query and stream processing.

Distributed file systems were addressed, as these are a common requirement for the retention of data in big data systems. HDFS was given emphasis, as it is the distributed file system of Hadoop and was used in this study. YARN was also covered, as it is the distributed resource manager of Hadoop and was also used.

Shell scripting and command pipelining were addressed, as these were central to the development with Unicage and to the system administration, deployment and configuration activities required to maintain the experimental clusters used for this study.

Big data systems were addressed with emphasis on Hadoop, Hive, Spark and Unicage, as these are noteworthy systems spanning three distinct data processing paradigms: Hadoop and Hive enable batch and query processing over MapReduce; Spark enables the same modalities over RDDs; and Unicage enables the same modalities through shell scripting.

Chapter 3

Related Work

This chapter discusses previous works that inspired this study. Section 3.1 surveys the current big data benchmarking landscape. Section 3.2 addresses previous studies that also employed the use of Unicage. A summary of the chapter is provided in Section 3.3.

3.1 Big Data Benchmarks

The main goals of benchmarking consortia like the Transaction Processing Performance Council (TPC)[1] and the Standard Performance Evaluation Corporation (SPEC)[2] are to provide objective data regarding the performance and efficiency of computing systems, and to define standards for their benchmarking so as to prevent bias and favoring of specific tools. Big data benchmarks follow and expand on the standards defined by TPC and SPEC, and are classified based in their scope, in three categories [44]:

- *Micro-benchmarks*: benchmarks designed to evaluate individual system components through small operations;
- *End-to-end benchmarks*: benchmarks designed to evaluate whole systems through typical application scenarios;
- *Benchmark suites*: combinations of micro and end-to-end benchmarks targeted at various big data systems.

What follows is a survey on open-source benchmarks targeted at big data systems in cluster environments.

[1] http://tpc.org/
[2] https://spec.org/

3.1.1 BigDataBench

BigDataBench[3] [10, 11] is a big data benchmark suite, where the basic units of scope are *data motifs* - classes of small units of computation, such as a sorting operation [45]. A big data workload can be seen as a pipeline of data motifs performed on initial or intermediate data inputs. BigDataBench includes micro-benchmarks for singular data motifs, component benchmarks for combinations of data motifs, and end-to-end benchmarks to simulate application scenarios with combinations of component benchmarks. It provides a suite of implementations of batch, query and stream processing workloads for the state-of-the-art big data systems, including Apache Hadoop, Hive and Spark, as well as a plethora of artificial inteligence workloads. It uses the Big Data Generation Suite (BDGS) [46] to generate input data sets for the benchmarks.

BDGS

Purposefully developed as a component of BigDataBench, BDGS is a data generator capable of producing large volumes of structured, semi-structured and unstructured synthetic data sets. BDGS addresses the properties of big data with the exception of velocity, since it is not suitable as a data ingestion tool for stream processing. It addresses the veracity of big data by respecting the realistic semantic properties of the input samples it uses to generate data sets with simulated value.

A data generation workload with BDGS is characterized by a raw data sample from which the generated data set derives, an input seed that introduces variance, and a desired volume. To generate structured data, BDGS borrows the capabilities of the Parallel Data Generation Framework (PDGF) [47].

PDGF

Developed at the University of Passau, PDGF is a system capable of parallel and distributed generation of structured data. There are two core releases: the *demo* release is restricted to data generation in a single machine, while the *full* release extends the demo with multi-node distributed data generation.

A data generation workload with PDGF is characterized by the *runtime configuration file*, an XML file that determines various aspects of the generation workload, such as the degree of parallelism and number of nodes; and by the *data schema configuration file*, an XML file that specifies the input seed and the schema of the structured data set to be generated.

[3] https://benchcouncil.org/BigDataBench/

3.1.2 BigBench

BigBench is an end-to-end benchmark targeted at data warehousing and query processing frameworks. It was first introduced in 2013 [48] and later adapted into an industry standard by TPC, the TPCx-BB[4] benchmark. It uses query processing workloads and data sets recreating those of a product retailer. BigBench extends PDGF to generate structured, semi-structured and unstructured data. In 2017, BigBench V2 [49] was proposed as a rework of the original data model and data generator to better reflect real-life scenarios.

3.1.3 HiBench

HiBench[5] is a benchmark suite proposed in 2010 [50]. Its latest release provides implementations for batch, query and stream processing workloads, as well as machine learning and processing of graph data with a variety of big data systems, including Apache Hadoop, Spark, Flink, Storm and Kafka.

3.1.4 AMPLab Big Data Benchmark

AMPLab Big Data Benchmark[6] is a collection of micro-benchmarks targeted at data warehousing and query processing frameworks, such as Hive and Spark SQL. It provides implementations of the *select*, *join* and *aggregation* workloads, as proposed by Pavlo et al. [51] for the evaluation of query processing frameworks.

3.1.5 SparkBench

SparkBench[7] was proposed in 2015 [52] and is a benchmark suite targeted specifically at Apache Spark. It provides a suite of implementations for query and stream processing workloads, as well as machine learning and processing of graph data. The data generators of SparkBench are implemented as Spark workloads.

3.1.6 Comparing the Big Data Benchmarks

The aforementioned big data benchmarks are targeted at complex big data systems and do not address lean systems. Table 3.1 presents a comparison between all surveyed big data

[4] http://tpc.org/tpcx-bb/default5.asp, https://github.com/h2oai/tpcx-bb
[5] https://github.com/Intel-bigdata/HiBench/
[6] https://amplab.cs.berkeley.edu/benchmark/
[7] https://github.com/CODAIT/spark-bench

benchmarks, sorted by the order of appearance in the section. The list of supported big data systems is not exhaustive.

		Targeted big data systems			
Big data benchmark	Scope	Batch processing	Query processing	Stream processing	Data generator
BigDataBench	Suite	Hadoop, Spark, Flink	Hive, Spark	Spark, Flink	BDGS
HiBench	Suite	Hadoop, Spark	Spark	Spark, Storm, Flink	HiBench
BigBench (TPCx-BB)	End-to-end	n/a	Hive, Spark	n/a	PDGF
AMPLab Big Data Benchmark	Micro	n/a	Hive, Spark	n/a	HiBench
SparkBench	Suite	n/a	Spark	Spark	Spark

Table 3.1: Comparison between big data benchmarks.

The study of these benchmarks reveals common requirements for big data benchmarking, including mechanisms to generate input data sets with controllable volumes, and a set of workloads implemented in a set of big data systems. The *grep*, *sort*, *wordcount*, *select*, *join* and *aggregation* workloads are commonly featured among the batch and query processing benchmarks. Spark, as a framework suitable for most workload types, is the most benchmarked big data system.

3.2 Unicage Studies

This section details previous studies that employed the use of Unicage.

3.2.1 LeanBench

Moreira et al. [8] benchmarked the performance of Unicage Tukubai, Hadoop and Hive, deployed and configured in a single machine. The benchmarks addressed the *grep*, *sort* and *wordcount* batch processing workloads; and the *select*, *join* and *aggregation* query processing workloads. The captured metrics were the execution time, the data processed per second and the usage of system resources.

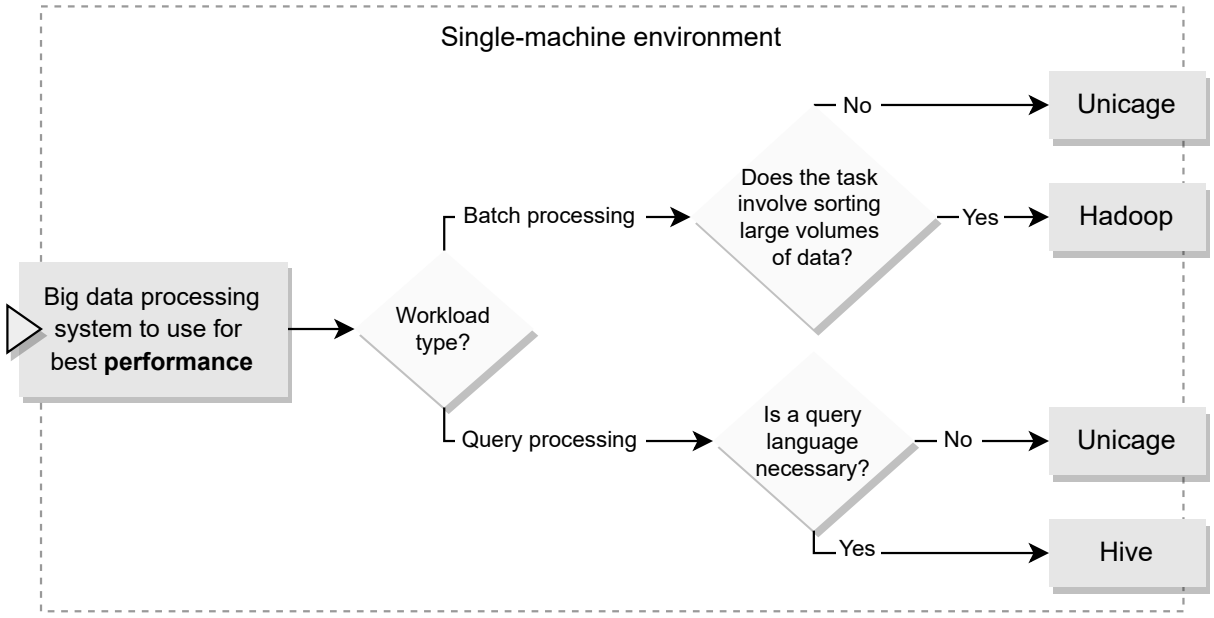


Figure 3.1: Summary of the conclusions of LeanBench.

Figure 3.1 highlights the conclusions of LeanBench: in a single-machine environment, Unicage performs better than Hadoop if the task does not involve sorting large data sets (over 400 MB), and performs better than Hive for data sets of all volumes, with the latter being the preferred system only if using a query language is a requirement. These results suggest that the complexity of the software stack might be a limitative factor when it comes to the performance of big data systems.

The main limitation of LeanBench was the deployment in a single-machine environment, which prevented tests with larger data sets.

3.2.2 Unicage and Smart Grids

Recent work by Ferreira et al. [53] demonstrated a use-case with Unicage Tukubai applied to the processing of data generated from smart meters from a same smart grid. These devices monitor the electricity, water and gas consumption of a population and produce large volumes of data in the form of XML files. Unicage can be used to convert these XML files into smaller text files, preserving the business value of the data and reducing the storage burden of storing XML files directly. A 27 GB data set with 27 million files containing smart meter readings was manually partitioned in batches of 1 million files and processed sequentially. The workload was characterized by the parsing, validation and aggregation of the smart meter data and took approximately 6 minutes to complete.

The main limitation of this demonstration was the deployment in a single-machine environment, which, like in LeanBench, prevented tests with larger data sets. Another limitation was the impractical partition of the used data set, as the partitioning strategy could have allowed the parallelization of the workload.

3.3 Chapter Summary

This chapter introduced previous studies in the field of big data benchmarking, all of which, with the exception of LeanBench, did not address lean big data systems. All covered big data benchmarks have mechanisms to generate input data sets. BDGS and PDGF were given emphasis, as they are suitable for the generation of unstructured and structured data, respectively. Among the most commonly benchmarked big data processing workloads were the *grep*, *sort* and *wordcount* batch processing workloads; and the *select*, *join* and *aggregation* query processing workloads. These serve distinct purposes and are the type of small operations that characterize micro-benchmarks.

Previous studies using Unicage were covered to highlight its capabilities as a lean big data system, but were limited to its deployment in a single machine, which prevented tests with larger data sets.

Our study re-used BDGS and PDGF for data generation, and expanded on the existing work with benchmarks for the aforementioned batch and query processing workloads in both complex and lean big data systems deployed in a cluster environment.

Chapter 4

Implementation

This study aimed to compare the performance of complex and lean big data systems with respect to batch and query processing workloads deployed in a cloud cluster environment. This chapter details the provisioning and configuration of the machines in the experimental environment, and the development, deployment and execution of the software required to run the set of benchmarking experiments entailed by the following research questions:

- **RQ 1**: *In a cloud computing cluster setting, how do complex and lean big data systems compare from a performance perspective?*
- **RQ 2**: *In a cloud computing cluster setting, how do complex and lean big data systems handle the volume of big data?*
- **RQ 3**: *In a cloud computing cluster setting, how do complex and lean big data systems handle the variety of big data?*

An overview of the typical big data benchmarking experiment is presented in Section 4.1. The infrastructure isolates the Hadoop and Unicage environments in two identical data processing sub-clusters: the *Hadoop cluster* and the *Unicage cluster*. The provisioning and configuration of these clusters are described in Section 4.2, along with the remaining infrastructure. The benchmarked workloads are listed and described in detail in Section 4.3. A summary of the chapter is provided in Section 4.4.

4.1 Experimental Overview

All experiments followed a similar execution plan. Figure 4.1 represents the structure of a typical benchmarking experiment in a data-flow diagram, where a process represents a step in

the experiment, trailed by its corresponding output. The generation of the input data sets for the benchmarks is represented by the process called *data generation*, and is described in Section 4.1.1. The two important benchmarked processes are *data loading* and *data processing*, and these are described in Section 4.1.2 and Section 4.1.3, respectively. The collection of metrics pertaining to the execution of the workloads is represented by the process called *monitoring*, and is described in Section 4.1.4. The output of the data processing task (the *output* file) is subject to further scrutiny to deem the benchmark results (the *benchmark results* file) reliable or unreliable. This is represented by the process called *verification*, and is described in Section 4.1.5. The statistical validation of the results is addressed in Section 4.1.6.

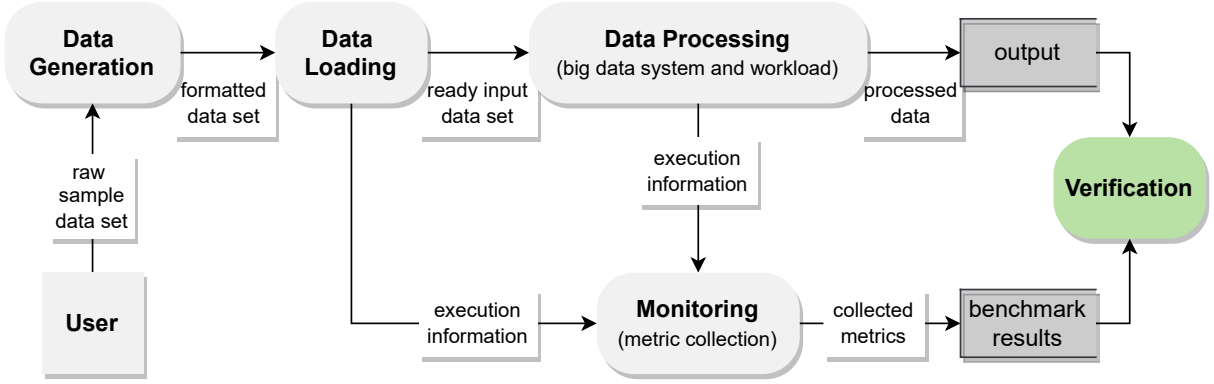


Figure 4.1: Data-flow diagram of a big data benchmarking experiment.

4.1.1 Data Generation

The following input data sets were generated for the benchmarks:

- *Wikipedia text entries*: unstructured data in the form of artificial text built from a set of 7776 unique words;
- *E-commerce tables*: structured data in the form of two tables - the *item* and the *order* tables - and their schema definitions.

The data generators were deployed in a *Producer cluster* (described in Section 4.2), and the data generation workloads were parallelized and distributed equally among the *Producer* nodes. Figure 4.2 shows the average generation times with four configured nodes in the Producer cluster. The vertical axis shows the execution time, in hours, and the horizontal axis shows the planned volumes of the data sets, in gigabytes.

The unstructured data sets simulating Wikipedia text entries were generated with BDGS in batches of files with a fixed size of about 500 MB each. The starting timestamp was used as

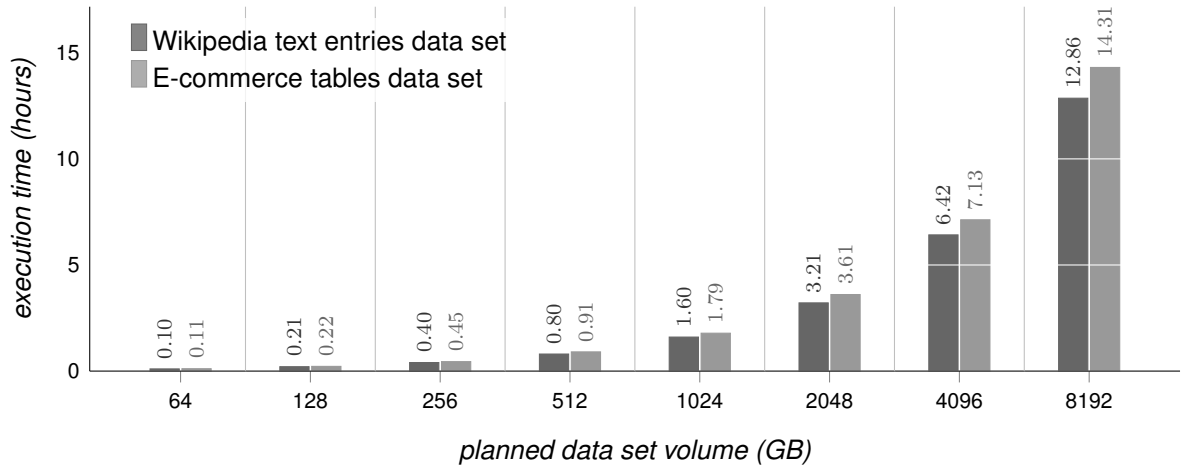


Figure 4.2: Average data generation times with four Producer nodes.

input seed. Each Producer node was in charge of generating its corresponding fraction of the desired volume. For these data sets, any sufficient increase in volume resulted in an increase of the number of generated files.

The structured data sets simulating e-commerce tables were generated with PDGF and were based in a provided schema (described in detail in Section 4.3.2). A pseudo-random value generated from the `/dev/urandom`[1] file was used as input seed. The full release of PDGF was not available, so the demo release was purposefully extended to allow multi-node distributed data generation - each Producer node was in charge of generating its corresponding fraction of rows for each table, which resulted in one file per table in each Producer. For these data sets, any increase in volume resulted in an increase of the size of each generated file.

The data generators for structured and unstructured data did not generate data sets with the exact planned volume. Table 4.1 lists the real generated data set volumes and associated percent deviation, in relation to the planned data set volumes.

	Planned volumes (GB)							
Data set	64	128	256	512	1024	2048	4096	8192
Wikipedia text entries	62.68 (-2.06%)	125.25 (-2.15%)	250,66 (-2.09%)	501,22 (-2.11%)	1002,18 (-2.13%)	2004,77 (-2.11%)	4008.97 (-2.13%)	8018.78 (-2.11%)
E-commerce tables	68.16 (+6,50%)	138.84 (+8.47%)	280.79 (+9,68%)	564.68 (+10.29%)	1144.61 (+11.78%)	2320.33 (+13.30%)	4671.93 (+14,06%)	9416.87 (+14.95%)

Table 4.1: Data generation: the real generated data set volumes and percent deviation in relation to the planned data set volumes.

[1] **/dev/urandom**: a special file present in UNIX systems that enables pseudo-random number generation through readings of environmental noise data from device drivers.

Each table of the structured e-commerce tables data sets had a fraction of the total volume. Table 4.2 lists the real generated volumes of the *item* and *order* tables of the e-commerce tables data sets.

	Real volumes (GB)							
Data set	68.16	138.84	280.79	564.68	1144.61	2320.33	4671.93	9416.87
E-commerce *item* table	42.50	85.84	172.71	346.45	698.00	1406.42	2823.43	5671.80
E-commerce *order* table	25.66	53.00	108.08	218.23	446.61	913.91	1848.50	3745.07

Table 4.2: Data generation: the real generated table volumes of the e-commerce tables structured data sets.

4.1.2 Data Loading

Identical input data sets were loaded separately from the Producer cluster into the Hadoop and Unicage clusters. This was a benchmarked process, where the minimum possible set of operations required to load the data into each cluster was strictly considered.

Loading into the Hadoop Cluster

The input data sets were loaded into the Hadoop cluster by having the nodes in the Producer cluster act as HDFS Clients capable of loading data into HDFS using the `put` operation. The Producer nodes had no influence in the execution of the data processing workloads, since they were not configured as NameNodes or DataNodes. The partitioning and distribution of the input data sets across the DataNodes was automatically achieved by HDFS. Listing 4.1 shows how a file is loaded into the Hadoop cluster.

```
producer:$ hadoop fs -put data.txt /hdfs/directory/
```

Listing 4.1: Shell CLI: Loading a file into the Hadoop cluster.

Loading structured data into the Hadoop cluster required the additional step of creating the corresponding Hive tables, a process performed by the NameNode after the input data set was loaded into HDFS. If the `data.txt` data set loaded in Listing 4.1 has a structured format, a Hive table can be created, as shown in Listing 4.2. Listing 4.3 shows the command used to execute the statement in Listing 4.2, and the `-f` flag preceeds the indication of the file the statement is written on.

```
1 -- Create the table and attribute schema
2 create table name_of_table(clmn1 int, clmn2 double, clmn3 string)
3 -- Indicate the delimiter used to separate columns in the raw file
4 row format delimited fields terminated by '|'
5 -- Indicate the path to the raw file in HDFS
6 stored as textfile location '/hdfs/directory/data.txt';
```

Listing 4.2: HiveQL statement: creating a Hive table based on a file stored in HDFS.

```
namenode:$ hive -f table_creation_statement.sql
```

Listing 4.3: Shell CLI: Running a statement in Hive.

Loading into the Unicage Cluster

Loading the input data sets into the Unicage cluster was achieved through shell scripting with the `distr-distr` command from the Unicage BOA library. The process was the same for loading both the structured and unstructured data sets. When a Producer node was finished generating its fraction of the input data set, the `distr-distr` command was used in that same Producer to equally partition and distribute its generated fraction across the specified Unicage worker nodes. Listing 4.4 shows how a file is loaded into the Unicage cluster.

```
producer:$ distr-distr $workers data.txt /worker/directory/data.txt
```

Listing 4.4: Shell CLI: Loading a file into the Unicage cluster.

The `distr-distr` command splits the specified file in a number of chunks equal to the number of specified Unicage workers, with a *round-robin* row distribution, and then forwards each chunk to the corresponding worker, where it is placed in a specified directory. In Listing 4.4, the `data.txt` file is split in chunks, and each chunk is placed in the `/worker/directory/` directory of each worker. The `$workers` variable contains a list of the addresses of the workers in the Unicage cluster.

4.1.3 Data Processing

After the data loading process, both the Hadoop and Unicage clusters had identical data sets and were prepared for the data processing benchmarks. Each data processing workload ran

thrice to allow the results to be statistically validated, as described in Section 4.1.6. Before each run, there was a clean-up process to free the resources of every node in the cluster. Table 4.3 lists the benchmarked batch and query processing workloads, and the tested big data systems that supported them. Each workload is described in detail in Section 4.3.

	Batch processing			Query processing		
	grep	*sort*	*wordcount*	*select*	*join*	*aggregation*
Hadoop MapReduce	Yes	Yes	Yes	No	No	No
Hive (over MapReduce)	No	No	No	Yes	Yes	Yes
Spark	Yes	Yes	Yes	Yes	Yes	Yes
Unicage	Yes	Yes	Yes	Yes	Yes	Yes

Table 4.3: Benchmarked workloads and big data systems.

4.1.4 Monitoring

The benchmarks were characterized by the following performance metrics:

- *Execution time*: the amount of time it took, in seconds, for the workload to finish executing;
- *Data loaded/processed per second*: also referred to as *loading/processing rate*, results from dividing the real size of the input data set by the execution time.

Additionally, the following resource usage metrics were collected to provide, when needed, a detailed view of the burden of each workload in the various nodes of the infrastructure, at any point during the execution:

- *CPU usage*: the percentage of CPU in use;
- *Memory usage*: the percentage of memory in use;
- *Disk I/O usage*: the input (read) and output (write) data transfer rates between the physical disk and memory;
- *Network throughput*: the network input and output volumes.

Netdata[2] was used to collect the resource usage metrics. Netdata is a tool that allows infrastructure monitoring and observability. A Netdata metric collector agent was hooked to each individual machine in the Hadoop and Unicage clusters. Upon the completion of a workload, the records pertaining to the timeframe of its execution were queried from Netdata into CSV

[2]https://www.netdata.cloud/

files and curated for analysis. Given the size and number of the collected resource usage metric plots, these are only reported in this document when relevant.

4.1.5 Verification

To assure the correctness of the benchmarks, the outputs resulting from each combination of big data system and workload were subject to a verification process.

A *cryptographic hash function* converts data of arbitrary size into an array of bits with a fixed size, also refered to as a *message-digest*, hereafter addressed as a *hash*. A cryptographic hash function is *deterministic*, meaning it produces the same hash for two equal inputs, and is *collision resistant*, meaning it is unlikely that two different inputs result in the same hash. Furthermore, any change made to the input data, even if minimalistic, results in a completely different hash, a trait called *avalanche effect*. These properties are fundamental in the context of output verification. Message Digest 5 (MD5) [54] meets these properties and produces a 128-bit hash that can be calculated, for a specified file, with the `md5sum` command in Linux. This command was used to produce the hashes from the outputs of the big data systems.

Each big data system produced the outputs in slightly different formats, so these had to be *homogenized* to allow the hashes to highlight disagreements. This process was different for each workloads and is shown in detail in Section 4.3.

4.1.6 Validation

Every benchmark was repeated thrice, and the sampled execution times and loading or processing rates were statistically validated through the estimation of the population mean and confidence intervals, as described by Montgomery et al. [55]. For any given benchmark, the population is considered to be a finite but very large number of repetitions of the same benchmark in the same environment. This process is described in detail in Appendix A.

Benchmarks whose individual runs were predicted to take more than 18 hours to finish were not addressed, in the interest of time.

4.2 Experimental Environment - Provisioning and Configuration

The experimental environment was comprised of a cluster, as illustrated in Figure 4.3. Every machine was deployed as a Virtual Server Instance (VSI) in the IBM Virtual Private Cloud (VPC) infrastructure, with the Ubuntu 20.04 operating system and kernel version 5.4.0-1015.ibm, and

was hosted in the same geographic location. The experimental cluster was divided in two data processing sub-clusters and one data generating sub-cluster:

- *Hadoop cluster*: for data processing with Hadoop, Hive and Spark;
- *Unicage cluster*: for data processing with Unicage shell script-based solutions;
- *Producer cluster*: for data generation and loading into the Hadoop and Unicage clusters.

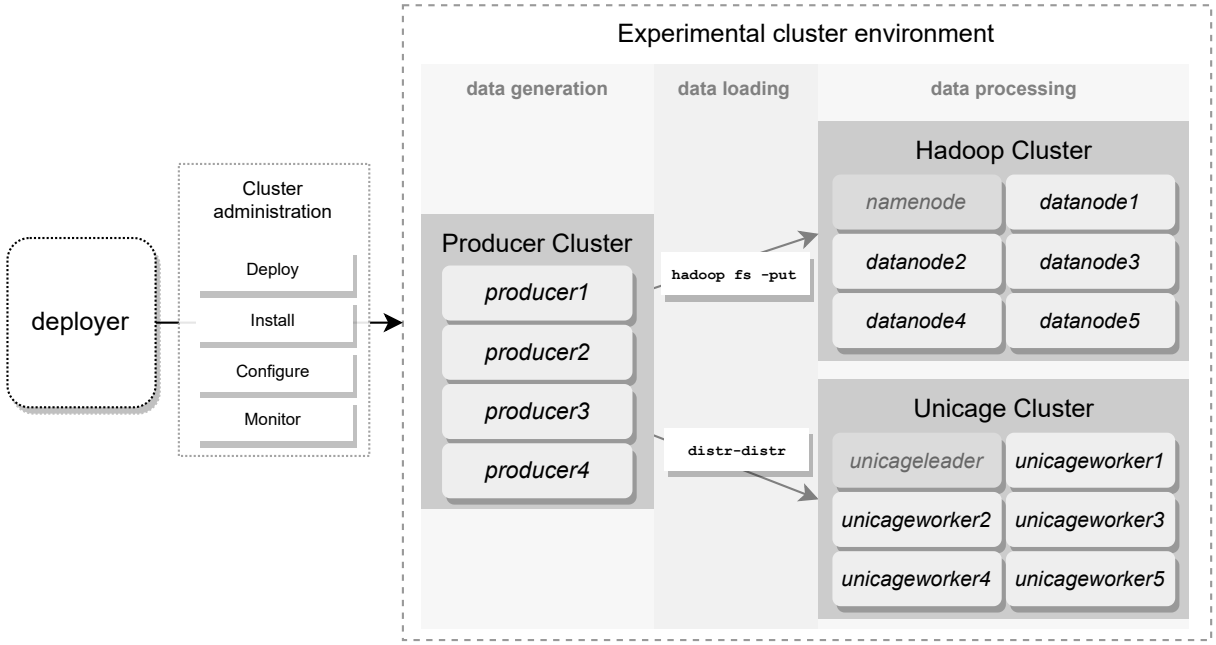


Figure 4.3: Architecture of the experimental cluster environment.

The Hadoop and Unicage clusters were identical in characteristics for an impartial benchmarking environment. Table 4.4 summarizes the characteristics of all nodes in the experimental cluster.

	Software				Hardware			
Hostname	Hadoop	Hive	Spark	Unicage	CPU	RAM	Network	Disk
Hadoop cluster								
namenode	3.3.1	3.1.2	3.2.1	n/a	8 vCPU, 2.0 GHz	32 GB	12 Gbps	1024 GB, 412 Mbps
datanode1...5	3.3.1	n/a	n/a	n/a	4 vCPU, 2.0 GHz	16 GB	6 Gbps	4096 GB, 1607 Mbps
Unicage cluster								
unicageleader	n/a	n/a	n/a	TKB, BOA	8 vCPU, 2.0 GHz	32 GB	12 Gbps	1024 GB, 412 Mbps
unicageworker1...5	n/a	n/a	n/a	TKB, BOA	4 vCPU, 2.0 GHz	16 GB	6 Gbps	4096 GB, 1607 Mbps
Producer cluster								
producer1...4	3.3.1	n/a	n/a	TKB, BOA	8 vCPU, 2.0 GHz	32 GB	12 Gbps	4096 GB, 1607 Mbps

Table 4.4: Specifications of the experimental cluster environment.

4.2.1 Hadoop Cluster

The Hadoop cluster had six nodes with a fully distributed installation of Hadoop 3.3.1, Hive 3.1.2 and Spark 3.2.1. The NameNode, whose hostname was *namenode*, had an Intel Xeon Cascadelake processor with 8 cores at 2.0 GHz, 32 GB of RAM, 12 Gbps of network bandwidth, and 1024 GB of disk storage with 3 IOPS/GB[3] and a maximum disk transfer rate of 412 Mbps. There were five DataNodes, whose hostnames ranged from *datanode1* to *datanode5*, each with an Intel Xeon Cascadelake processor with 4 cores at 2.0 GHz, 16 GB of RAM, 6 Gbps of network bandwidth, and 4096 GB of disk storage with 3 IOPS/GB and a maximum disk transfer rate of 1607 Mbps. HDFS was installed in the DataNodes for a total of 20480 GB of distributed storage.

4.2.2 Unicage Cluster

The Unicage cluster had six nodes with installations of the Unicage Tukubai and Unicage BOA libraries. The Unicage leader node, whose hostname was *unicageleader*, had an Intel Xeon Cascadelake processor with 8 cores at 2.0 GHz, 32 GB of RAM, 12 Gbps of network bandwidth, and 1024 GB of disk storage with 3 IOPS/GB and a maximum disk transfer rate of 412 Mbps. There were five Unicage worker nodes, whose hostnames ranged from *unicageworker1* to *unicageworker5*, each with an Intel Xeon Cascadelake processor with 4 cores at 2.0 GHz, 16 GB of RAM, 6 Gbps of network bandwidth, and 4096 GB of disk storage with 3 IOPS/GB and a maximum disk transfer rate of 1607 Mbps. The total storage capacity of the Unicage worker nodes was of 20480 GB.

4.2.3 Producer Cluster

The Producer cluster had four Producer nodes, whose hostnames ranged from *producer1* to *producer4*. BDGS and PDGF were installed on all four nodes for data generation. Hadoop was also installed on all four nodes despite them not being configured neither as NameNodes or DataNodes, but purely as HDFS Clients. This allowed data to be loaded into Hadoop cluster. The Tukubai and BOA commands required to load data into the Unicage cluster were also installed in the Producer cluster. Each Producer node had an Intel Xeon Cascadelake processor with 8 cores at 2.0 GHz, 32 GB of RAM, 12 Gbps of network bandwidth, and 4096 GB of disk storage with 3 IOPS/GB and a maximum disk transfer rate of 1607 Mbps.

[3] **IOPS/GB**: input/output operations per second, per gigabyte.

4.2.4 Cluster Administration

The deployment of all the needed tools in the cluster was automated with idempotent shell scripts that could also be used for the reconfiguration of the big data systems under test, if needed. These shell scripts were implemented to run from a central node with hostname *deployer*, as shown in Figure 4.3, with connection to all the other nodes. This node had all tools installed for convenience, but did not interfere in the benchmarks and had the minimal hardware required for simple cluster administration. A large storage volume was attached to this node to temporarily store benchmark results and metrics.

4.2.5 Additional Configurations

Additional configurations were required for the benchmarks to be as impartial as possible.

The benchmarks did not contemplate replication, so file block replication was disabled in the Hadoop cluster by changing the `dfs.replication` property to 1, in the configuration file of HDFS.

In HDFS, by default, the maximum number of blocks per file is of 10000, and since each block has 128 MB in size, the biggest file size permitted in the filesystem is 1280 GB. HDFS imposes this limit to avoid performance overheads caused by the increased seek time when there are excessive numbers of file blocks in the filesystem. In some experiments, the input data sets had files that exceeded 1280 GB in size. To prevent problems with the loading of such data sets into the Hadoop cluster, the maximum number of blocks per file was configured by changing the `dfs.namenode.fs-limits.max-blocks-per-file` property to a sufficiently large number, in the configuration file of HDFS. In our case, 11100 was a good value.

Hive was configured to use the MapReduce processing engine and Derby, the default database for the Metastore.

Spark was deployed on top of HDFS and YARN in cluster mode. By default, when Spark is used with YARN, it launches two Executors and each uses a single CPU core. With this default configuration, Spark only uses two DataNodes and does not take advantage of the hardware of the cluster. Considering the experimental cluster had five DataNodes, each with four CPU cores, Spark was configured to launch five Executors, each able to use four CPU cores. This was achieved by changing the `spark.executor.instances` property to 5 and the `spark.executor.cores` property to 4, in the configuration file of Spark.

The caching level of Spark was left with the default configuration: when Spark runs out of memory to cache RDD partitions, it uses the lineage information of the partitions to reconstruct them on demand, continuing the workload at reduced performance.

Spark was also configured to use the Metastore of Hive to access the tables and schema of the structured data sets.

4.3 Benchmarked Workloads

This section describes all benchmarked workloads in detail and ties them with the research questions. Combined, the benchmarks provide the means to gather comprehensive conclusions regarding research question **RQ 1**: *In a cloud computing cluster setting, how do complex and lean big data systems compare from a performance perspective?*

4.3.1 Batch Processing

The batch processing benchmarks addressed the *grep*, *sort* and *wordcount* workloads. The *grep* and *wordcount* workloads received, as input, the unstructured data sets simulating Wikipedia text entries. These data sets featured artificial text built from a set of 7776 unique words. The *sort* workload received, as input, an alternative version of the data sets, where all words were arranged in a single column.

The intensity of running batch processing workloads stems from the volume of the input data sets, making these benchmarks favorable to reach conclusions regarding research question **RQ 2**: *In a cloud computing cluster setting, how do complex and lean big data systems handle the volume of big data?*

Grep Workload

Grep is a *search workload*. It should return the number of times a given substring occurs in the input data set. The benchmarked implementations were programmed to search for the substring ”area”, as it appeared frequently in the input data sets, both as a standalone word and as a substring within words.

Figure 4.4 shows a sample input for *grep*, its expected outputs and their verification. Figures like these are recurrent throughout this section and highlight the expected outputs of the workloads, and how these outputs are *homogenized* to be verified through the comparison of their hashes. In Figure 4.4, the input data set contains three instances of the ”area” substring. The outputs are homogenized to have the count of the substring as the first field and the substring itself as the second field, with no additional characters. The MD5 hashes of the outputs are compared, and the verification process concludes all big data systems are in agreement.

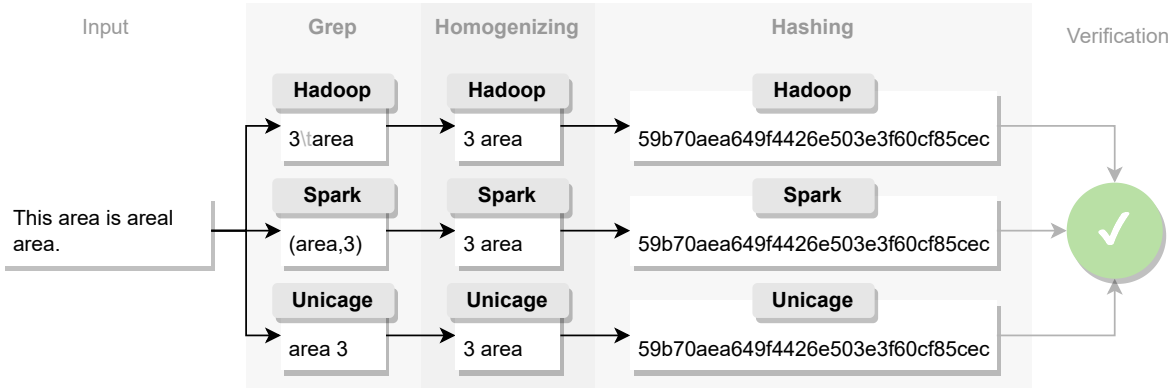


Figure 4.4: *Grep*: sample input data sets and expected outputs, followed by their verification.

Sort Workload

In the data sets used in the *sort* benchmarks, the words were separated by the newline character and arranged in a single column, resulting in about 1.6×10^8 rows per GB. The *sort* workload should return the sorted column of words. Figure 4.5 shows a sample input for *sort*, its expected outputs and their verification.

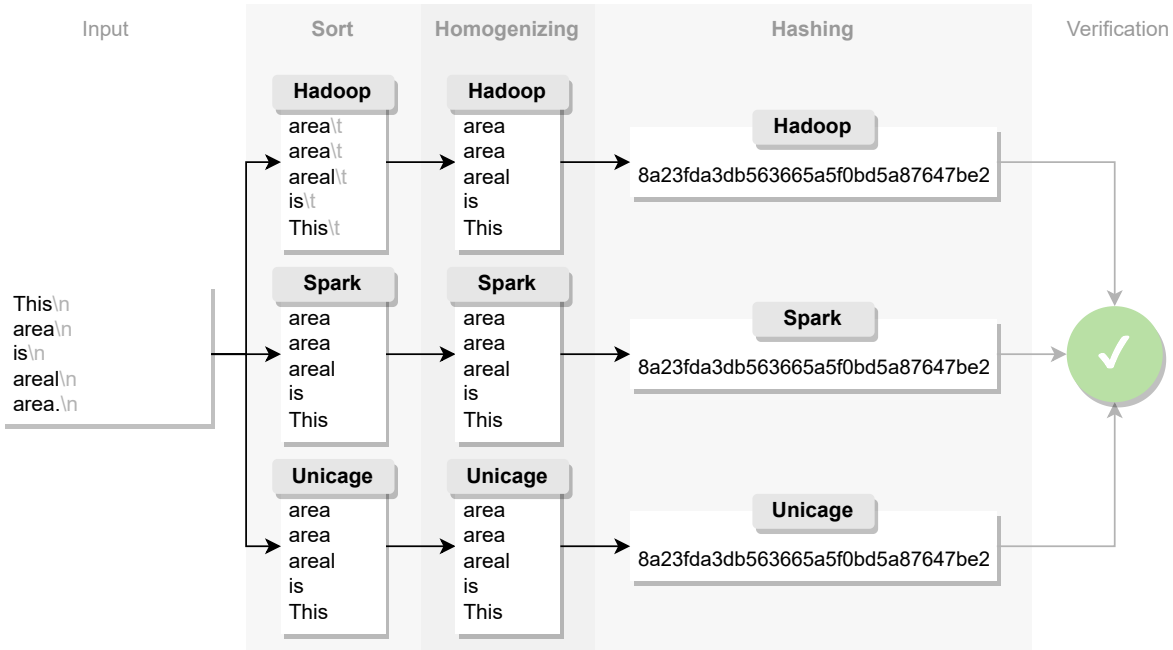


Figure 4.5: *Sort*: sample input data sets and expected outputs, followed by their verification.

Wordcount Workload

Wordcount is a *grouping workload*. It should return a list of all the words present in the input data set, together with the number of occurrences of each word. Figure 4.6 shows a sample

input for *wordcount*, its expected outputs and their verification.

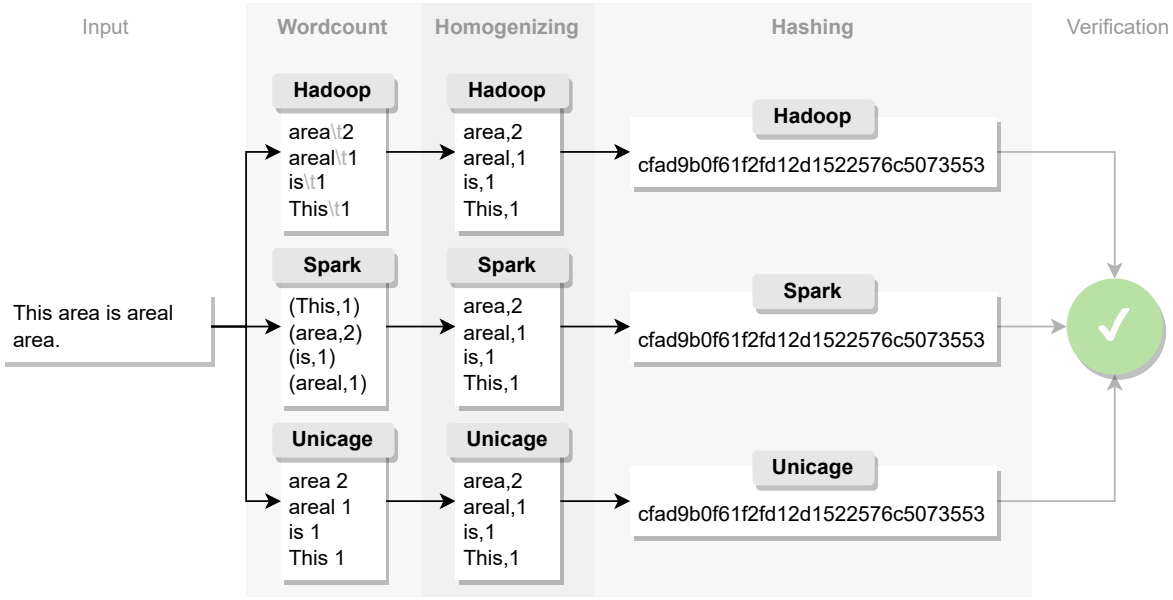


Figure 4.6: *Wordcount*: sample input data sets and expected outputs, followed by their verification.

4.3.2 Query Processing

The query processing benchmarks addressed the *select*, *join* and *aggregation* workloads. All workloads received, as input, the structured data sets simulating e-commerce tables, as shown in Figure 4.7, where the format of the sample corresponds to that of the data generators - the fields in each row are separated by the pipe character. In the statement previously shown in Listing 4.2 to create Hive tables from a file in HDFS, the fields were also separated like this.

Table: `ecom_order`

Schema:

`order_id` (int)
min = 0, unique
`buyer_id` (int)
min = 0, unique
`create_data` (string)
date format

Sample:

11100|33300|2007-04-24
11101|33301|2011-05-06
11102|33302|2011-06-15
11103|33303|2010-08-16

Table: `ecom_item`

Schema:

`item_id` (int)
min = 0, unique
`order_id` (int)
min = 0, permutation of ecom_order.order_id
`goods_id` (int)
min = 0, max = 999999
`goods_number` (int)
min = 0, max = 999
`goods_price` (double)
min = 0.01, max = 1000.00
`goods_amount` (double)
goods_number * goods_price

Sample:

0|42594|3020|567|661.40|375117.12
1|11102|1276|601|275.27|165442.89
2|11101|3020|992|998.43|990442.56
3|11102|1267|995|999.01|994014.95

Figure 4.7: Schema and sample structured input data set for the query processing benchmarks.

Like with batch processing, the intensity of running query processing workloads stems from the volume of the input data sets, making these benchmarks favorable to further investigate research question **RQ 2**. The batch processing benchmarks used unstructured data sets, while the query processing benchmarks used structured data sets. This dichotomy was favorable to reach conclusions regarding research question **RQ 3**: *In a cloud computing cluster setting, how do complex and lean big data systems handle the variety of big data?*

Select Workload

Like *grep*, *select* is a search workload. It should return the rows in the `ecom_item` table, whose `goods_amount` field is larger than 990000. In the benchmarked implementations, only the `goods_price` and `goods_amount` fields of the resulting table were output. Listing 4.5 implements this behavior with a SQL query. Figure 4.8 shows the expected outputs of the *select* workload and their verification. This and the analogous figures for the remaining query processing workloads use the sample data set in Figure 4.7 as input.

```
1 create table select_result as
2 select goods_price, goods_amount from ecom_item
3 where goods_amount > 990000;
```

Listing 4.5: SQL query: the benchmarked *select* workload.

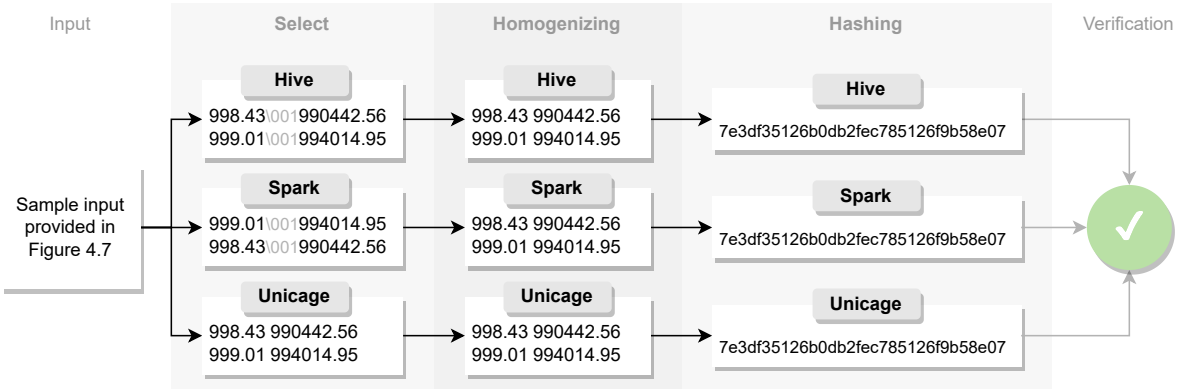


Figure 4.8: *Select*: expected outputs followed by their verification.

Join Workload

The *join* workload should return the table that combines the `ecom_item` and `ecom_order` tables, where the `order_id` field is matched. In the benchmarked implementations, only the

`buyer_id` and `goods_amount` fields of the resulting table were output. Listing 4.6 implements this behavior with a SQL query. Figure 4.9 shows the expected outputs of the *join* workload and their verification, applied to the sample input in Figure 4.7.

```
1 create table join_result as
2 select ecom_order.buyer_id, ecom_item.goods_amount from ecom_item
3 join ecom_order on ecom_item.order_id = ecom_order.order_id;
```

Listing 4.6: SQL query: the benchmarked *join* workload.

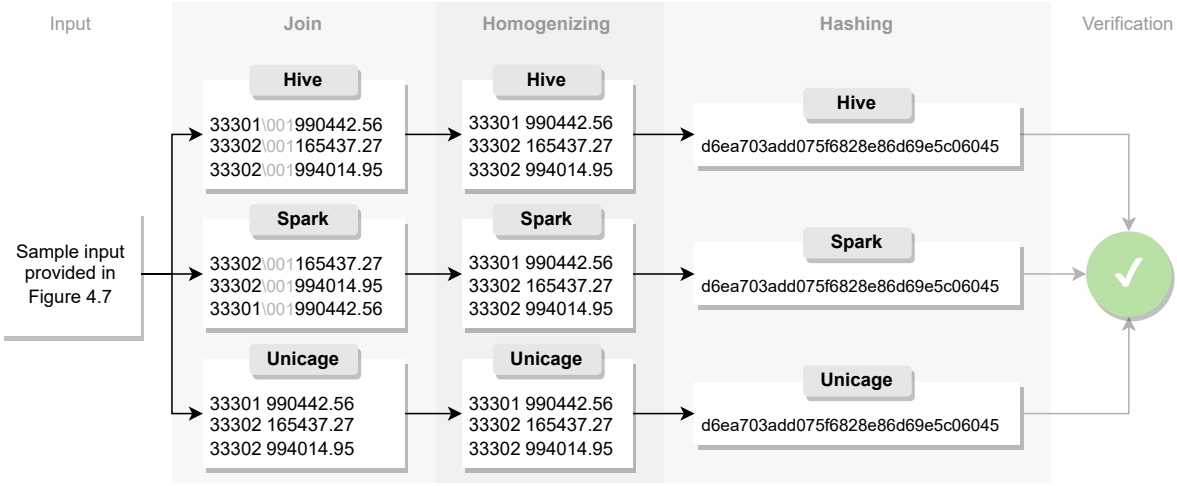


Figure 4.9: *Join*: expected outputs followed by their verification.

Aggregation Workload

Like *wordcount*, *aggregation* is a grouping workload. It should return, for the `ecom_item` table, all unique values in the `goods_id` field and the sums of the corresponding values of the `goods_number` field. Listing 4.7 implements this behavior with a SQL query. Figure 4.10 shows the expected outputs of the *aggregation* workload and their verification, applied to the sample input in Figure 4.7.

```
1 create table aggregation_result as
2 select goods_id, sum(goods_number) from ecom_item
3 group by goods_id;
```

Listing 4.7: SQL query: the benchmarked *aggregation* workload.

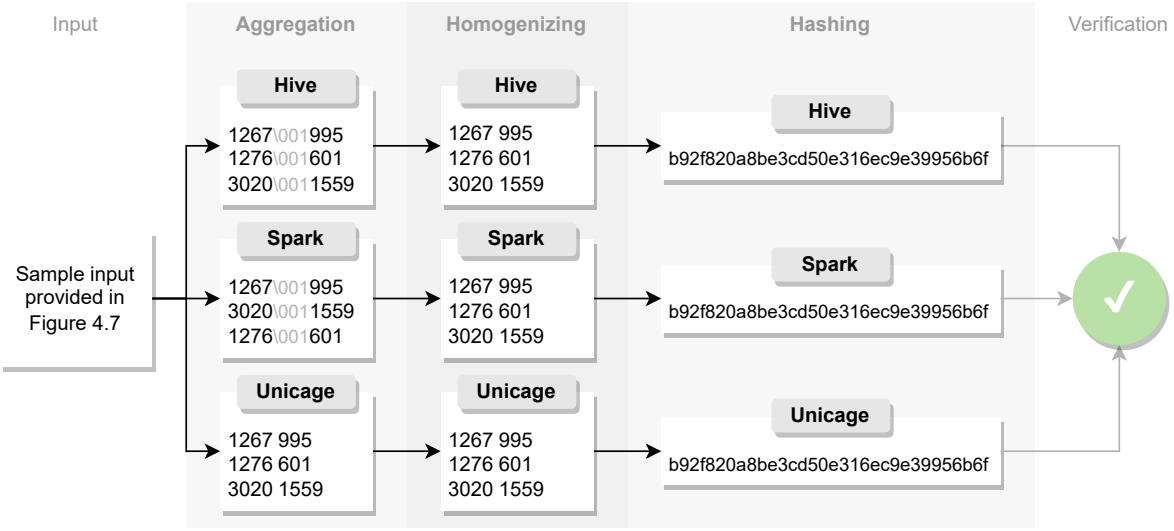


Figure 4.10: *Aggregation*: expected outputs followed by their verification.

4.3.3 The Benchmarked Workloads in Real-World Scenarios

All of the aforenamed workloads are small operations commonly used for micro-benchmarks, but figure prominently in real-world scenarios. To visualize this, let us think of the general example of sensors installed in a given country.

The use of batch or query processing workloads depends on the type of data produced by the sensors, with the former being suitable for unstructured data and the latter suitable for structured data.

Search workloads like *grep* and *select* filter data that meets specified conditions. This type of workloads could be used, for instance, to find the sensors whose measurements exceed a certain legal limit, prompting targeted action.

Grouping workloads like *wordcount* and *aggregation* group data based on specified patterns. This type of workloads could be used, for instance, to know the total measurements registered by the sensors of each city. This could prompt municipal responses for cities whose total measurements are not within legal boundaries.

The *sort* workload could be used, for instance, to know which sensors registered the highest monthly measurements in the country, prompting prioritized action.

Let us consider that there are two types of sensors associated with different activities, each producing structured data in the form of their own tables. The *join* workload could be used to find correlations between the two activities.

4.4 Chapter Summary

This chapter described the steps of the big data benchmarking experiments conducted to address the research questions of this study. These questions focus on how complex and lean big data systems perform in cluster environments, and on how each system handles the volumes and varieties of big data.

The experimental environment had three sub-clusters: the Hadoop cluster, to benchmark the workloads in complex big data systems; the Unicage cluster, which was identical to the Hadoop cluster but was used to benchmark the lean implementations of the workloads; and the Producer cluster, where the input data sets were generated. The tested big data systems were configured to adjust to the benchmarking environment and to make the best use of the available resources. These configurations were covered thoroughly.

Unstructured data sets simulating Wikipedia text entries were generated with BDGS and used as input for the batch processing benchmarks. Structured data sets simulating e-commerce tables were generated with PDFS and used for the query processing benchmarks. These data sets were loaded into the Hadoop cluster using the `put` operation of HDFS, and into the Unicage cluster using the `distr-distr` Unicage command. Both processes were benchmarked.

The data processing benchmarks were divided in two sets: the batch processing set, addressing the *grep*, *sort* and *wordcount* workloads implemented with Hadoop, Spark and Unicage; and the query processing set, addressing the *select*, *join* and *aggregation* workloads implemented with Hive, Spark and Unicage. The workloads were monitored with the collection of performance metrics, including the execution times and loading or processing rates. Resource usage metrics were also collected with Netdata to aid the optimization of the workloads during their implementation. The outputs of any given workload were verified so as to highlight disagreements between the tested big data systems through the comparison of their respective MD5 hashes. All benchmarks were statistically validated with three repetitions.

Chapter 5

Experimental Evaluation

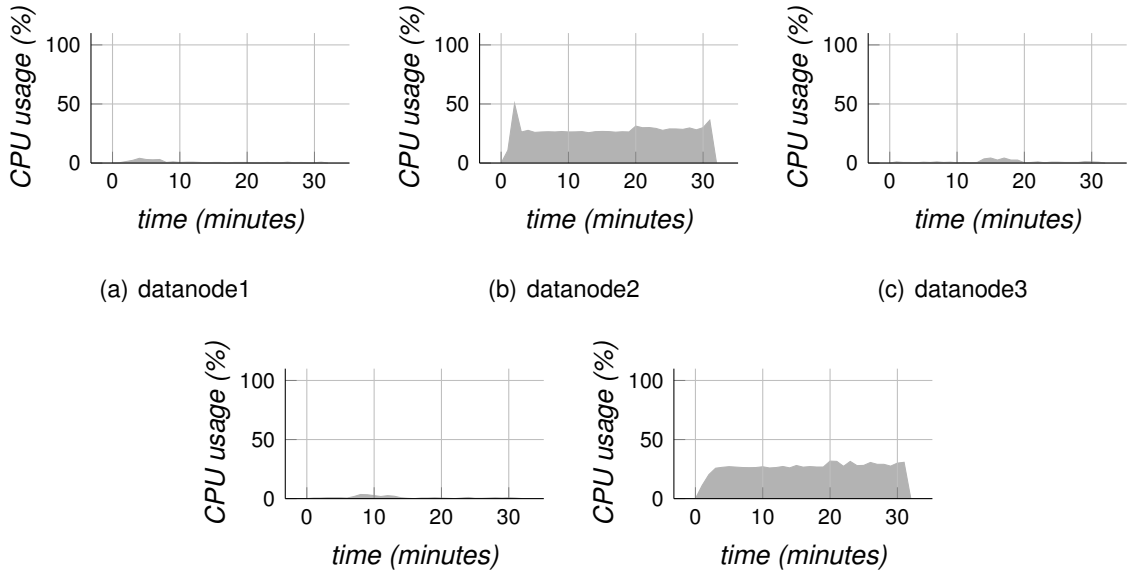
This chapter reports and discusses the results of the performance benchmarks. Preliminary results pertaining to the configurations performed to optimize the resource usage and execution times of the workloads are shown in Section 5.1. The results of the batch and query processing benchmarks are presented in Section 5.2 and Section 5.3, respectively. The reported benchmark results always reflect the sample mean values across three successful loading or processing runs, verified and validated as discussed in Sections 4.1.5 and 4.1.6 of Chapter 4. Section 5.4 provides additional discussion topics within the relevancy of the study. A summary of the chapter is provided in Section 5.5.

5.1 Preliminary

Preliminary experiments were necessary to learn the big data systems we set out to benchmark and to allow educated decisions regarding their optimization. These optimizations were necessary to make the best use of the resources available in the clusters. The following set of graphs provides evidence of the positive performance impact of these configurations.

Figure 5.1 shows the CPU usage of Spark executing a *wordcount* workload on a 125 GB data set, with the default configuration. Figure 5.2 shows the CPU usage of Spark executing the same workload with the same data set, but with the configuration described in Section 4.2.5 of Chapter 4. In the aforementioned figures, each graph pertains to one of the DataNodes in the Hadoop cluster. The vertical axis shows the total percentage of CPU activity, and the horizontal axis shows the execution time, in minutes.

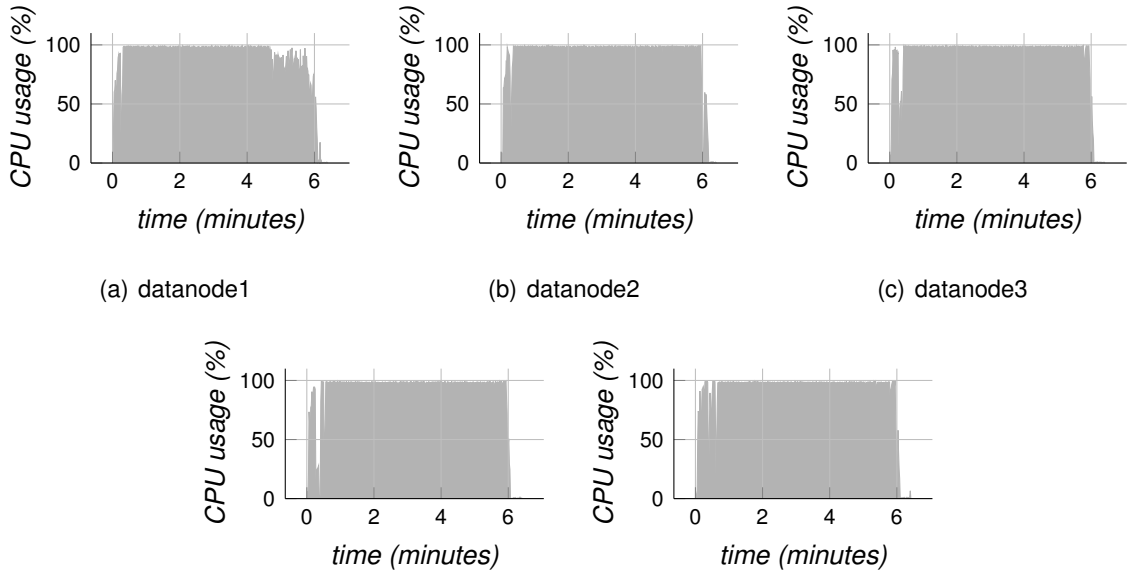
With the default configuration, Spark used a single CPU core in two of the DataNodes and took 32 minutes to complete the workload. With the alternative configuration, Spark used the four CPU cores available in each and every DataNode and took 6 minutes to finish.



(a) datanode1 (b) datanode2 (c) datanode3

(d) datanode4 (e) datanode5

Figure 5.1: DataNode CPU usage, when performing the *wordcount* workload on a 125 GB input data set with Spark: default configuration (2 Executors and 1 CPU core per Executor).



(a) datanode1 (b) datanode2 (c) datanode3

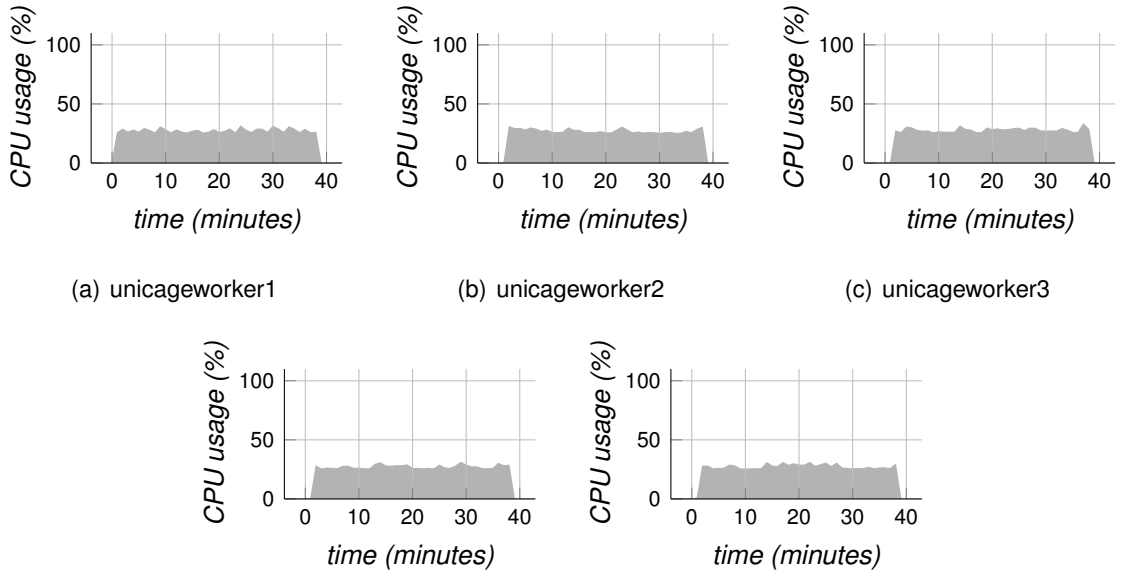
(d) datanode4 (e) datanode5

Figure 5.2: DataNodes CPU usage, when performing the *wordcount* workload on a 125 GB input data set with Spark: alternative configuration (5 Executors and 4 CPU cores per Executor).

The optimization of Unicage workloads was a manual endeavor that required the monitoring of the resource usage metrics to assert the quality of the implementations, which often times had to be parallelized to obtain the best performance. To show this, Figures 5.3 and 5.4 are analogous in presentation to Figures 5.1 and 5.2, but reflect the executions of a sequential and

of a parallelized implementations of *wordcount* with Unicage, using the same input data set.

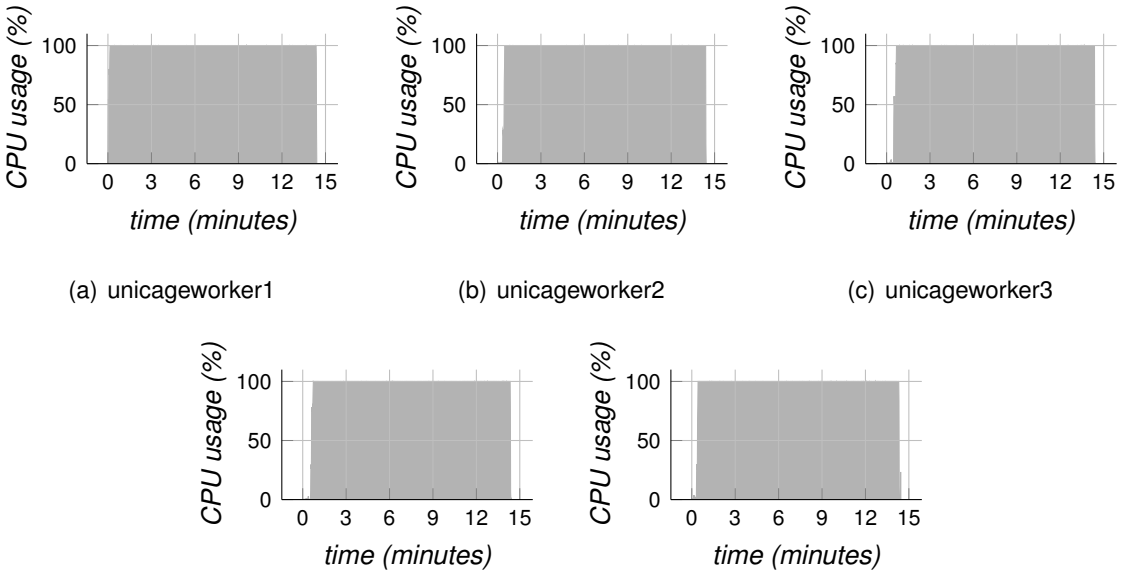
The sequential implementation used a single CPU core in each worker and took 38 minutes to execute. The parallelized implementation used a *first-in-first-out* thread scheduling logic to use all four CPU cores available in each worker and took 14 minutes to execute. This does not reflect the Unicage implementation of the *wordcount* workload used in the benchmarks.



(a) unicageworker1 (b) unicageworker2 (c) unicageworker3

(d) unicageworker4 (e) unicageworker5

Figure 5.3: Worker CPU usage, when performing the *wordcount* workload on a 125 GB input data set with a shell script-based Unicage solution: sequential implementation.



(a) unicageworker1 (b) unicageworker2 (c) unicageworker3

(d) unicageworker4 (e) unicageworker5

Figure 5.4: Worker CPU usage, when performing the *wordcount* workload on a 125 GB input data set with a shell script-based Unicage solution: parallelized implementation.

5.2 Batch Processing Benchmarks

The data sets generated and loaded into the data processing sub-clusters for the batch processing benchmarks were the Wikipedia text entries unstructured data sets. Figures 5.5 and 5.6 show the execution times and loading rates, respectively, of loading these data sets into both the Hadoop and Unicage clusters, for the tested volumes. These two types of bar charts are recurrent throughout this chapter.

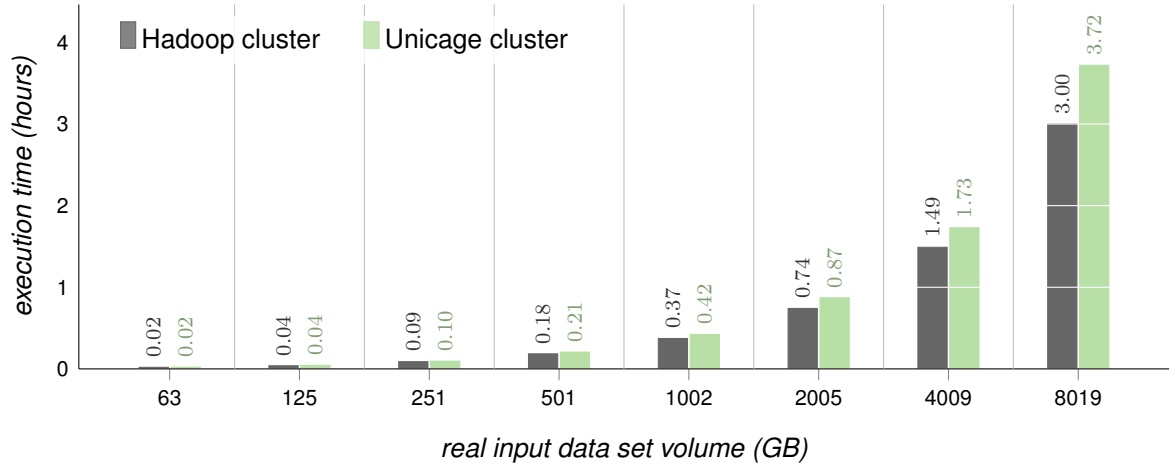


Figure 5.5: Data loading - Wikipedia text entries unstructured data sets: execution times.

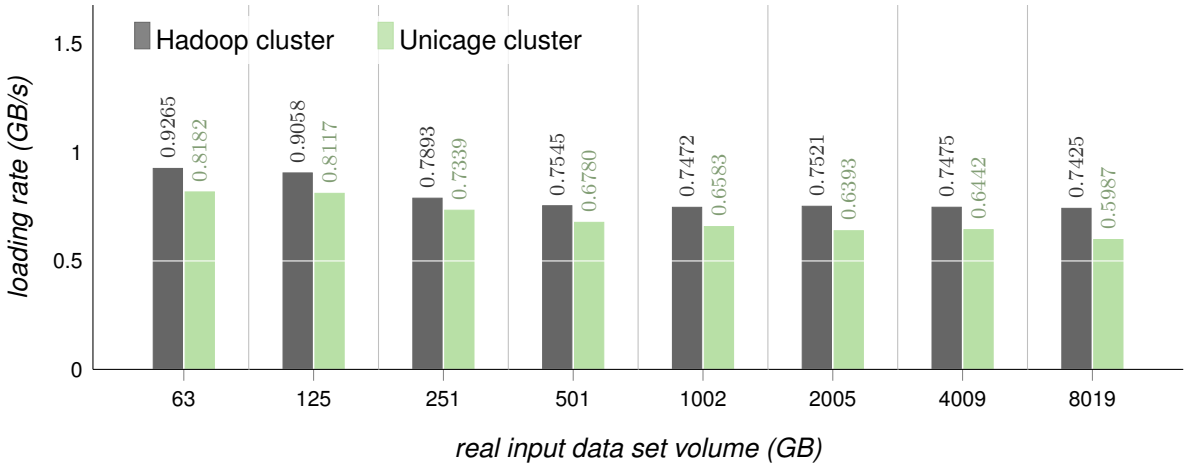


Figure 5.6: Data loading - Wikipedia text entries unstructured data sets: loading rates.

Bar charts pertaining to the *execution times* highlight the performance differences between the big data systems. In these charts, lower time bars mean better performance. The vertical axis shows the execution time, in hours, and the horizontal axis shows the set of tested input data set volumes, in gigabytes.

Bar charts pertaining to the *loading* or *processing rates* provide an understanding of the performance fluctuations of each big data system across the set of tested input data set vol-

umes. In these charts, higher rate bars mean better performance. The vertical axis shows the loading rate (for data loading workloads) or processing rate (for data processing workloads), in gigabytes per second, and the horizontal axis shows the set of tested input data set volumes, in gigabytes. In both types of chart, for easy reading, grey vertical lines mark the divisions between the data set volumes.

Loading the data into the Unicage cluster with the `distr-distr` command was slightly slower than loading it into the Hadoop cluster with the `put` operation of HDFS, across all tested volumes. The loading performance of Unicage had a small yet progressive degradation as the volumes increased (see the evolution of the Unicage bar in Figure 5.6). This was caused by the lookup, split and transfer latencies associated with the increase in the number of files of the data sets, which was proportional to the increase of the volumes - a 63 GB data set had 128 files, while a 8019 GB data set had 16380 files. This had a minimal impact in the loading performance of HDFS, as its loading rate plateaued for the 501 GB and larger data sets.

5.2.1 *Grep* Benchmarks

Figures 5.7 and 5.8 show the execution times and processing rates, respectively, of each big data system, when performing the *grep* workload for the tested input data set volumes.

The processing rate of Spark appears to be slower for the smaller data sets (compare the Spark bar for the 63 GB, 125 GB and 251 GB volumes in Figure 5.8). This was caused by its start-up period, which is more prominent when the workload itself takes little time to complete, such was the case of the *grep* and *select* workloads. This start-up period corresponds to when the SparkContext is created and the resources are allocated by YARN to the Spark Executors.

The processing rate of Spark fell significantly for the 8019 GB data set (compare the Spark bar for the 4009 GB and 8019 GB volumes in Figure 5.8). This is an indicative of the performance degradation Spark is known to experience when it runs out of memory in the cluster to cache RDD partitions [28]. A similar performance degradation was observed in the *select* and *aggregation* workloads.

Figure 5.9 shows the compound sum of the disk input rates of Spark, in the DataNodes, during the *grep* workload on the 4009 GB and 8019 GB data sets. The vertical axis shows the disk input rate, in mebibytes per second, and the horizontal axis shows the execution time, in minutes. The decline in disk input rate for the 8019 GB data set corresponds to when Spark stops retrieving data from disk into memory to cache the RDD partitions and starts recomputing these on demand.

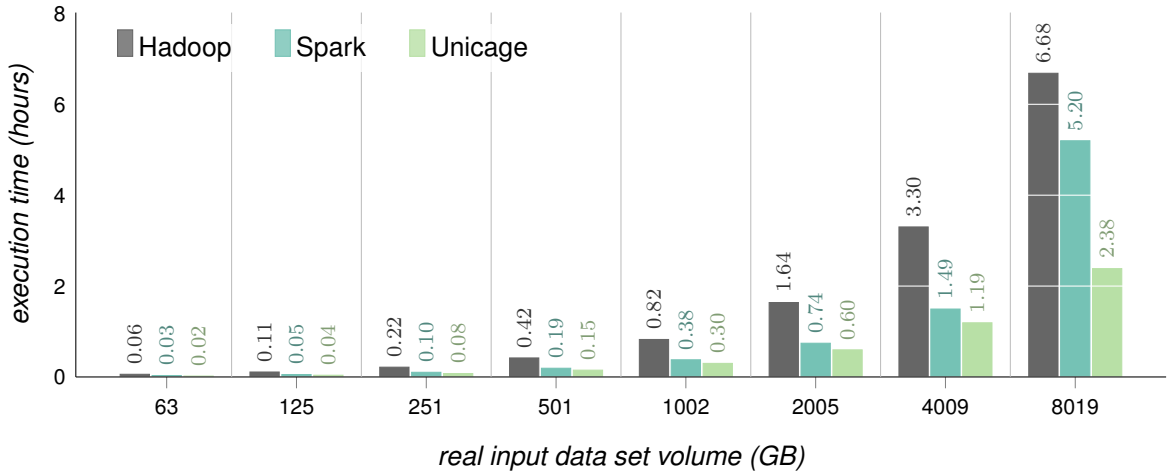


Figure 5.7: Data processing - *grep*: execution times.

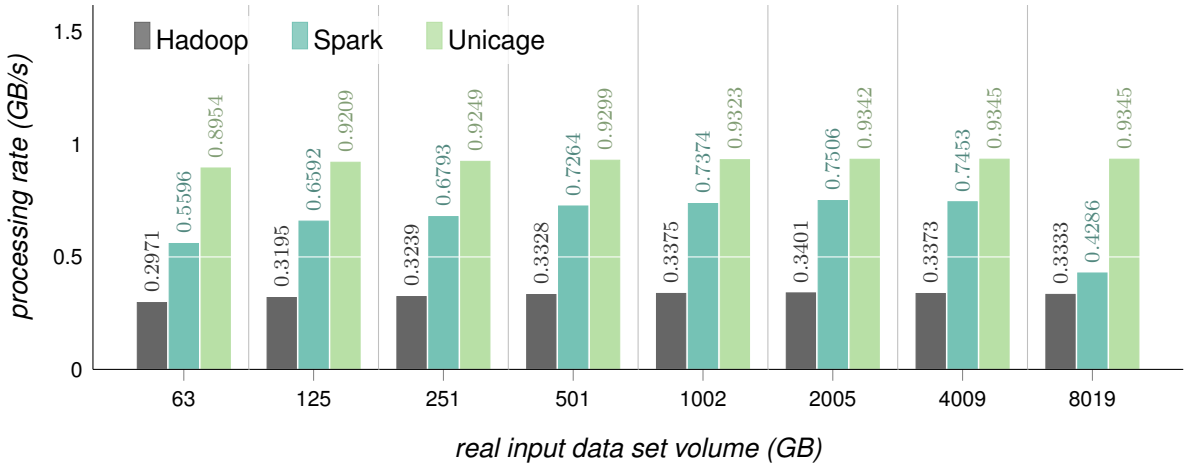
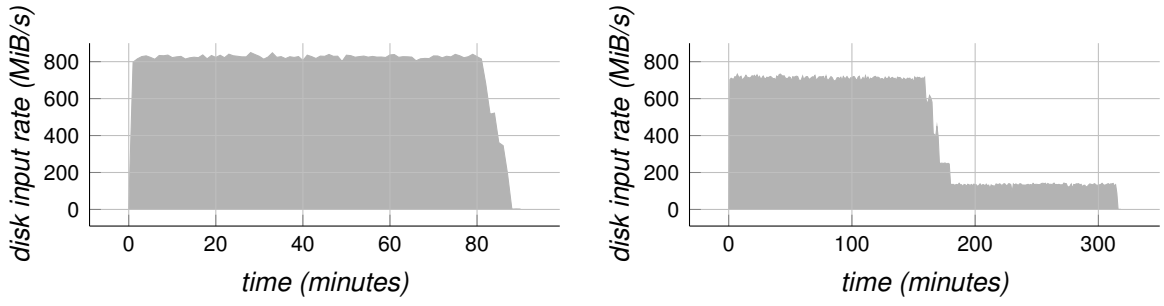


Figure 5.8: Data processing - *grep*: processing rates.



(a) 4009 GB input data set

(b) 8019 GB input data set

Figure 5.9: Compound sum of the disk input rates in the DataNodes, when performing the *grep* workload on the 4009 GB (a) and 8019 GB (b) input data sets with Spark.

In terms of completion, the *grep* benchmarks went without unpredictabilities. The Unicage implementation of the *grep* workload was faster than the Hadoop and Spark implementations, across all tested volumes.

5.2.2 *Sort* Benchmarks

Figures 5.10 and 5.11 show the execution times and processing rates, respectively, of each big data system, when performing the *sort* workload for the tested input data set volumes.

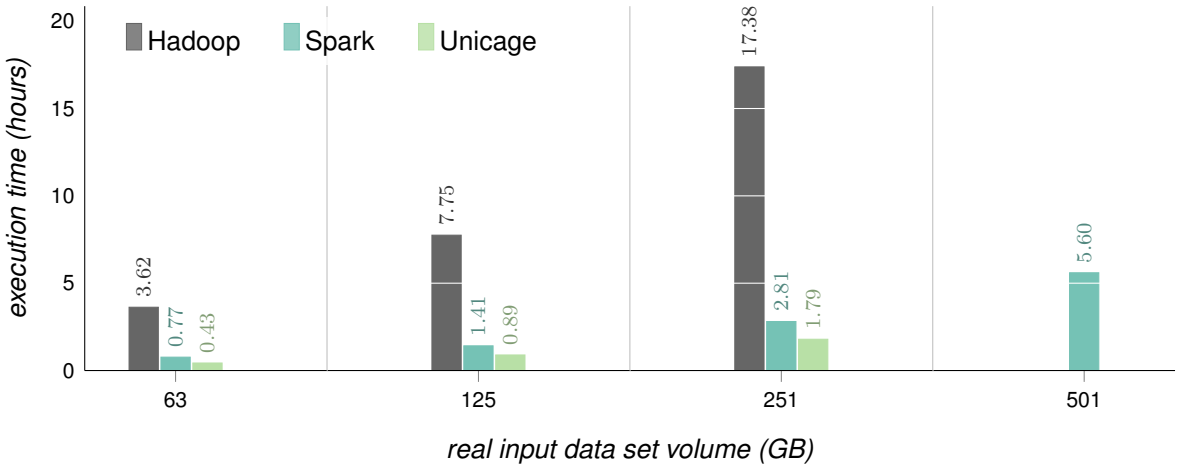


Figure 5.10: Data processing - *sort*: execution times.

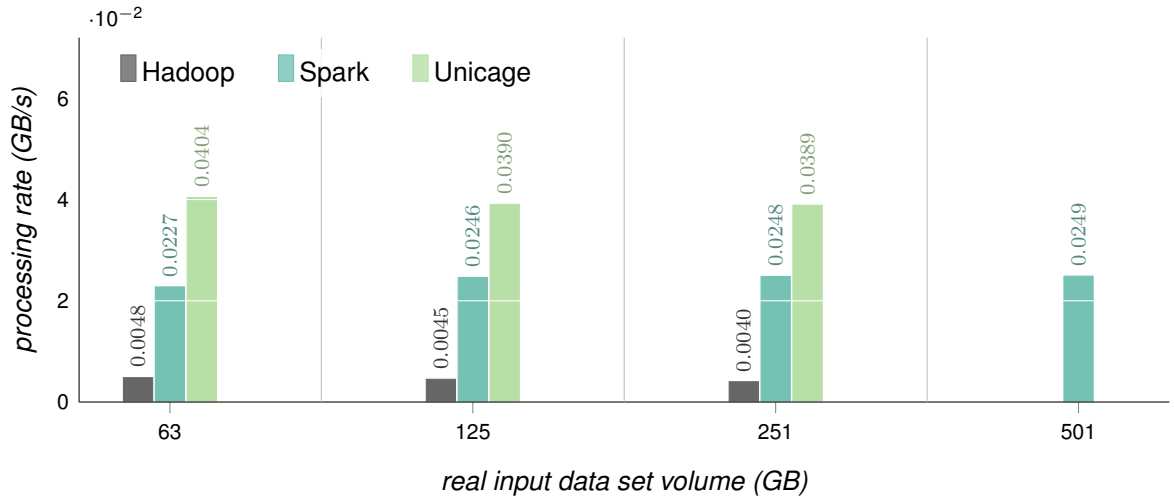


Figure 5.11: Data processing - *sort*: processing rates.

There are some missing results, as the *sort* benchmarks for the 1002 GB, 2005 GB, 4009 GB and 8019 GB data sets were not performed. The Unicage implementation produced arbitrarily incorrect outputs for the 501 GB data set. The execution time of sorting 1002 GB and larger data sets with Hadoop was predicted to widely exceed the imposed 18 hours limit, and thus there was no efficient way to verify the outputs of Spark for those data sets. The output of the Spark implementation for the 501 GB data set was verified through the comparison with the output of a single run of the Hadoop implementation, which took approximately 38 hours to complete, and thus was not validated statistically.

For the 63 GB, 125 GB and 251 GB data sets, the Unicage implementation of the *sort* workload was faster than the Hadoop and Spark implementations.

5.2.3 *Wordcount* Benchmarks

Figures 5.12 and 5.13 show the execution times and processing rates, respectively, of each big data system, when performing the *wordcount* workload for the tested input data set volumes.

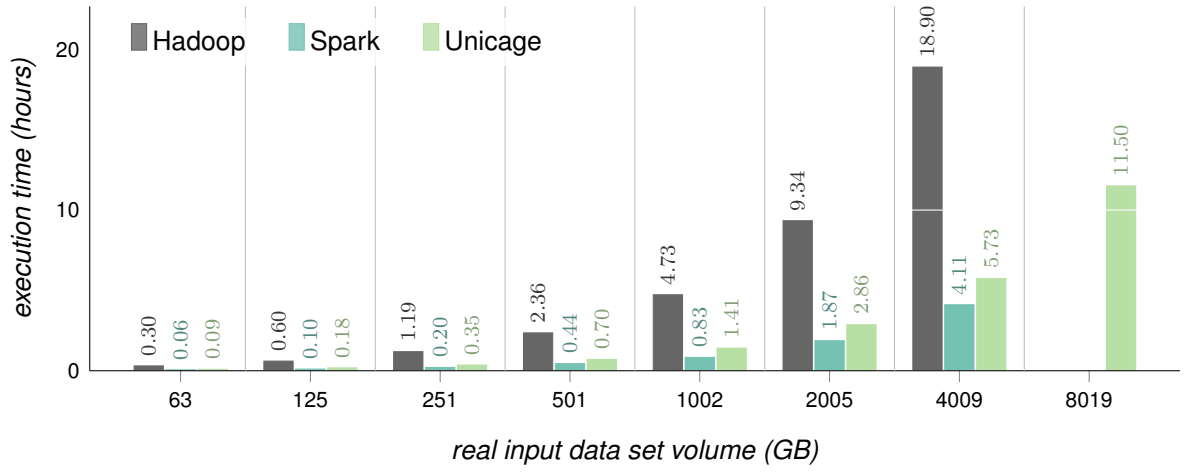


Figure 5.12: Data processing - *wordcount*: execution times.

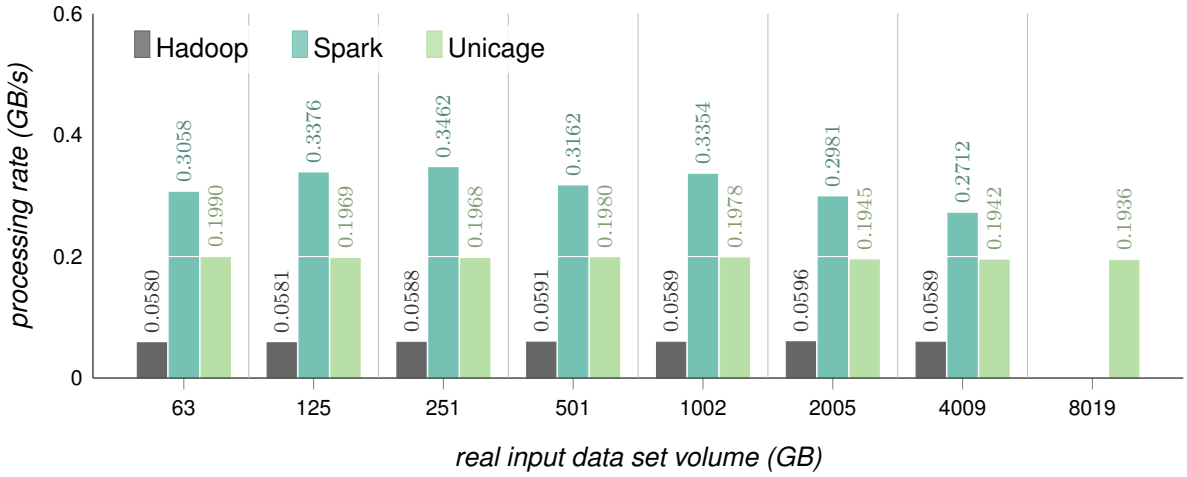


Figure 5.13: Data processing - *wordcount*: processing rates.

Spark was not able to complete the *wordcount* benchmark for the 8019 GB data set, as the JVM ran out of allocable space in the Java heap[1]. The Unicage implementation was able to finish this benchmark, and the results were verified through the comparison with the output of a single run of the Hadoop implementation, which took approximately 37 hours to complete, and thus was not validated statistically.

[1] **Java heap**: a dynamically allocated area of memory used to store objects instantiated by JVM applications.

For the data sets with volumes ranging from 63 GB to 4009 GB, the Unicage implementation of the *wordcount* workload was slower than the Spark implementation, but faster than the Hadoop implementation.

5.3 Query Processing Benchmarks

The data sets generated and loaded into the data processing sub-clusters for the query processing benchmarks were the e-commerce tables structured data sets. Figures 5.14 and 5.15 show the execution times and loading rates, respectively, of loading these data sets into both the Hadoop and Unicage clusters, for the tested volumes.

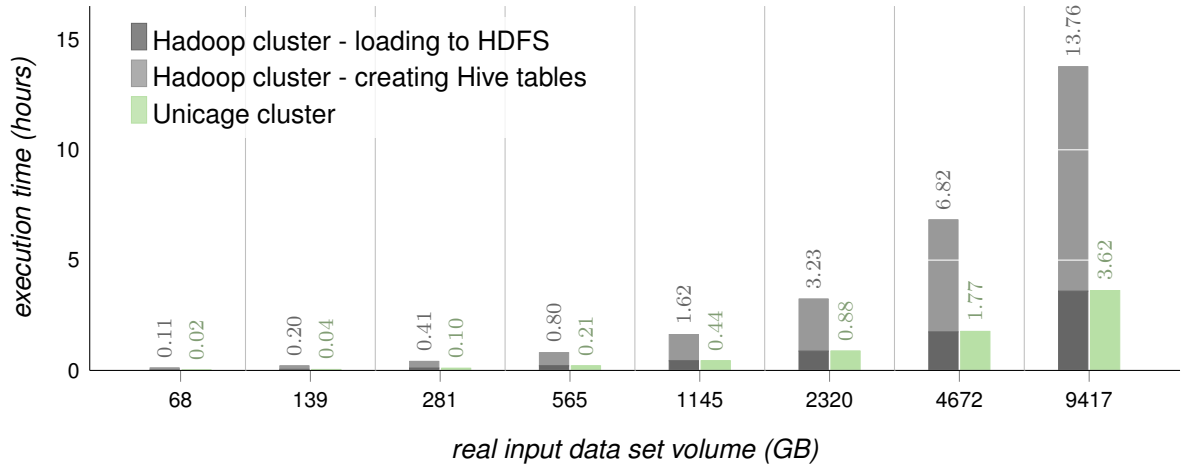


Figure 5.14: Data loading - e-commerce tables structured data sets: execution times.

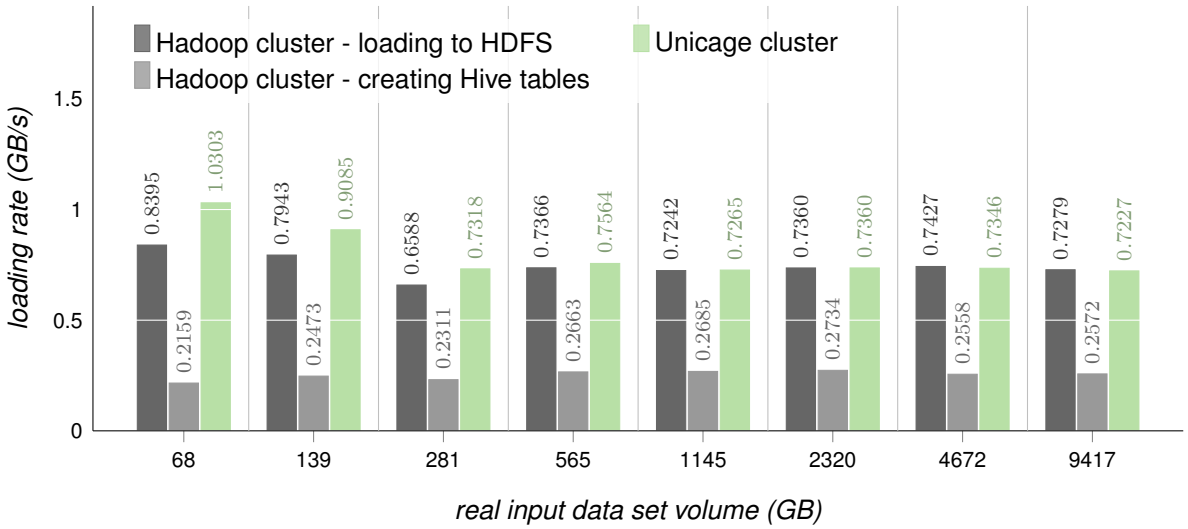


Figure 5.15: Data loading - e-commerce tables structured data sets: loading rates.

The stacked Hadoop bars in Figure 5.14 indicate that to load a structured data set into the Hadoop cluster, the step of loading the data into HDFS with the `put` operation was followed by the execution of the HiveQL statement to create the matching Hive tables. Unicage handles structured and unstructured data equally, so both were loaded into the Unicage cluster with the `distr-distr` command. This was significantly faster than the combination of steps required to load the structured data sets into the Hadoop cluster, across all tested volumes.

5.3.1 *Select* Benchmarks

Figures 5.16 and 5.17 show the execution times and processing rates, respectively, of each big data system, when performing the *select* workload for the tested input data set volumes. The *select* and *aggregation* workloads only used the *item* table of the e-commerce tables data sets, so the plotted volumes reflect the real volumes of that singular table.

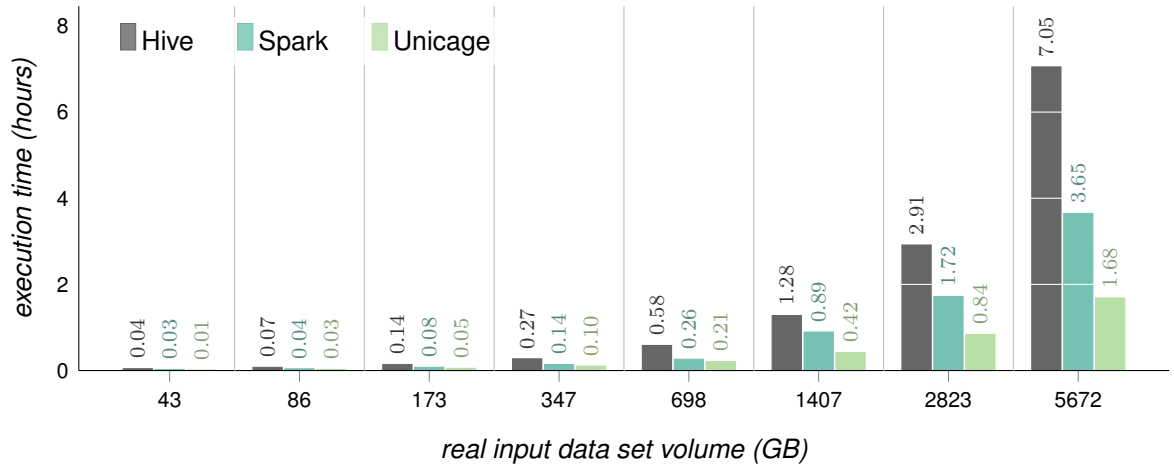


Figure 5.16: Data processing - *select*: execution times.

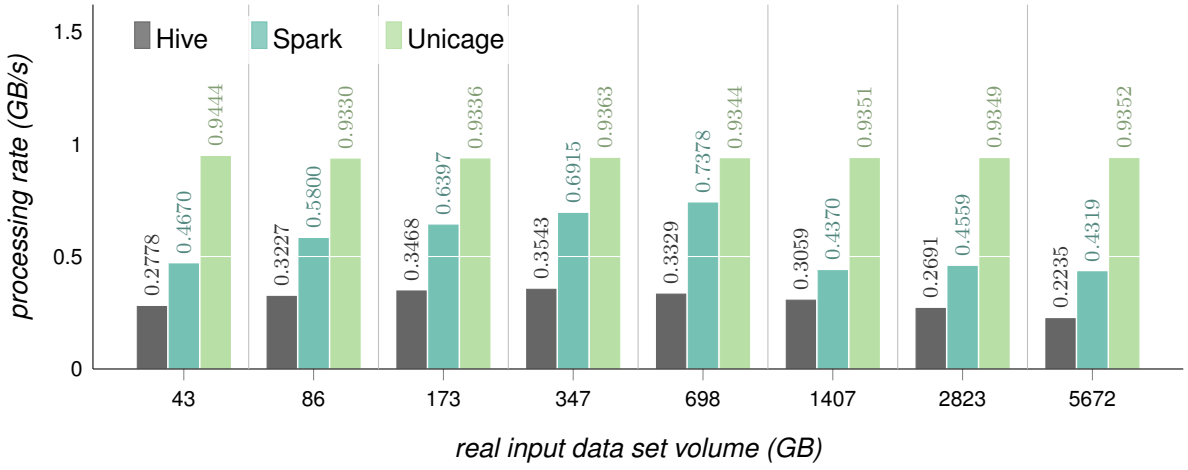


Figure 5.17: Data processing - *select*: processing rates.

The *select* benchmarks were completed without unpredictabilities. The Unicage implementation of the *select* workload was faster than the Hive and Spark implementations, across all tested volumes.

5.3.2 *Join* Benchmarks

Figures 5.18 and 5.19 show the execution times and processing rates, respectively, of each big data system, when performing the *join* workload for the tested input data set volumes.

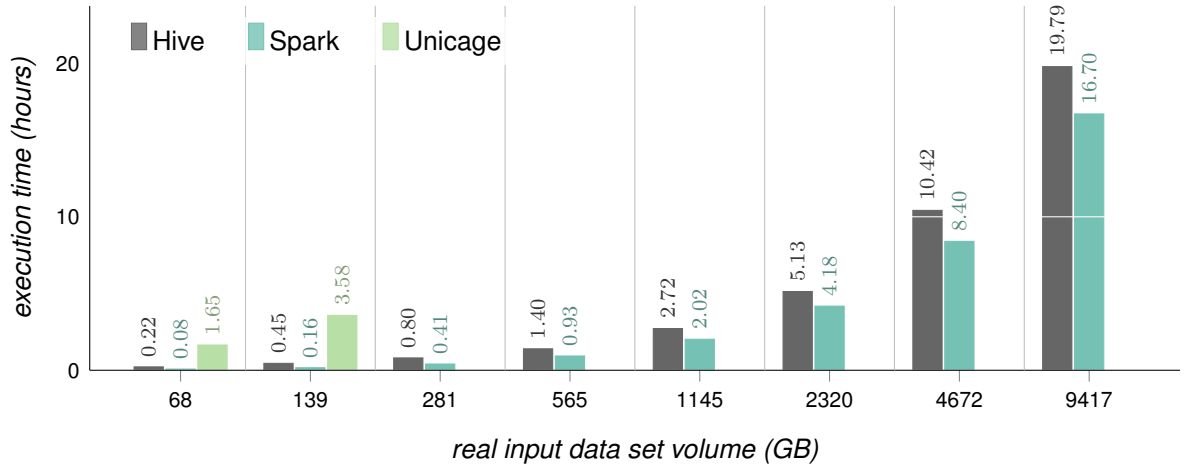


Figure 5.18: Data processing - *join*: execution times.

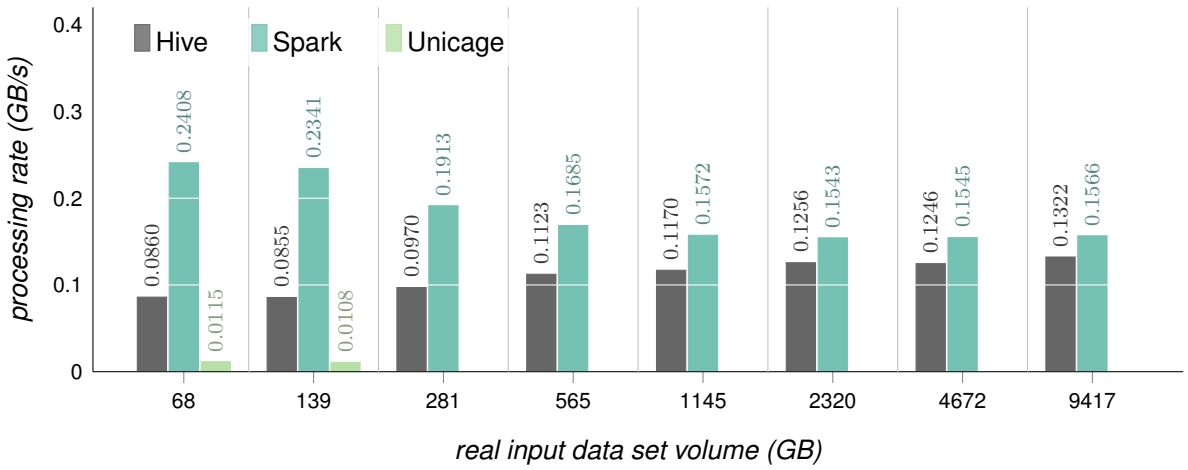


Figure 5.19: Data processing - *join*: processing rates.

The Unicage implementation of the *join* workload was largely inefficient from the start when compared to the Hive and Spark implementations (compare the Unicage bar with the remaining bars for the 68 GB and 139 GB volumes in Figures 5.18 and 5.19). To perform joins of rows of separate tables, Unicage requires these rows to be stored in the same node, so the Unicage cluster incurred in excessive latencies as a consequence of the significant transfer of large files from node to node. Unicage produced incorrect outputs for the 281 GB and larger data sets.

5.3.3 *Aggregation* Benchmarks

Figures 5.20 and 5.21 show the execution times and processing rates, respectively, of each big data system, when performing the *aggregation* workload for the tested input data set volumes.

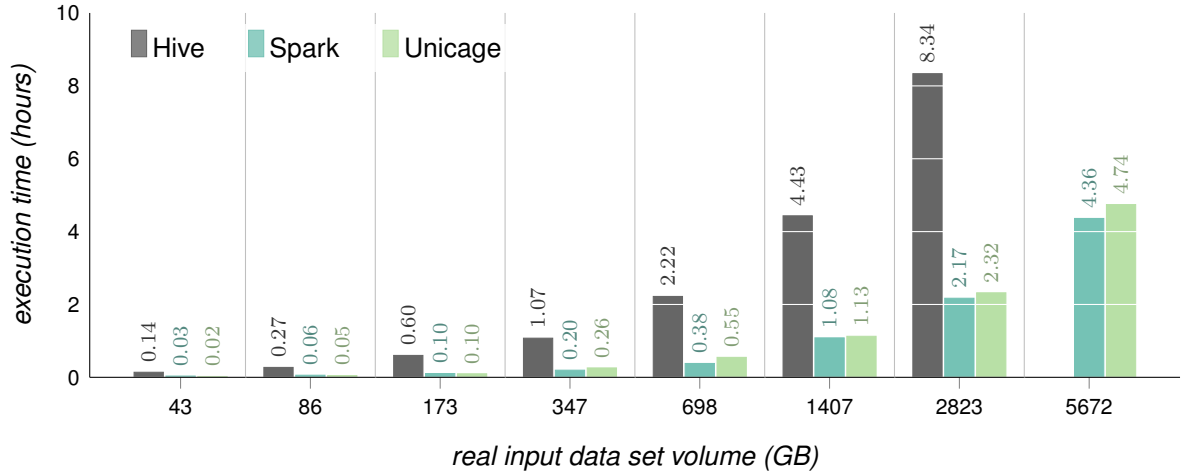


Figure 5.20: Data processing - *aggregation*: execution times.

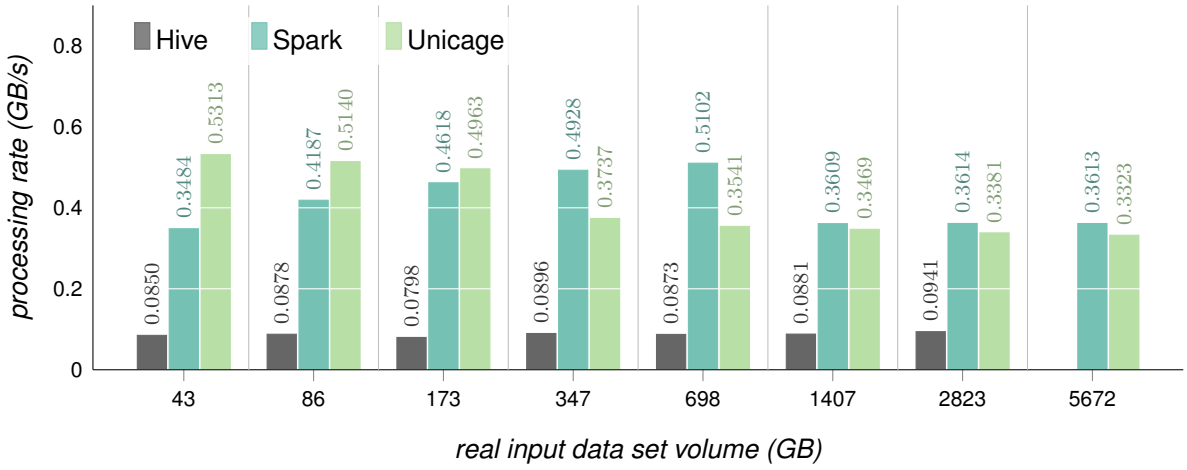


Figure 5.21: Data processing - *aggregation*: processing rates.

Hive was not able to perform the *aggregation* workload on the 5672 GB data set due to a failing MapReduce job. We did not further investigate this problem due to time constraints, and because Spark and Unicage managed to complete the workload significantly faster, with reciprocally verifiable results.

The *aggregation* benchmarks required two distinct Unicage implementations. The first was used for data sets up to 173 GB. The second was used for the 347 GB and larger data sets. The *aggregation* workload uses grouping operations to group data (e.g., the `group by` SQL operation and the `sm2` Unicage command). In Unicage, grouping operations require input data to be sorted, so the efficiency depends on the used sorting algorithm. The first Unicage imple-

mentation of the *aggregation* workload was faster than Spark (compare the Spark and Unicage bars for the 43 GB, 86 GB and 173 GB volumes in Figures 5.20 and 5.21), but it used the `msort` Tukubai command to perform an in-memory merge-sort, which proved to be problematic for the 347 GB data set, when the individual intermediate files became too big to be sorted in memory. The second Unicage implementation solved this problem by partitioning the intermediate files in chunks before the sorting phase. The performance impact of this additional step was captured in the drop observed in the processing rate of the Unicage implementation (compare the Unicage bar for the 173 GB and 347 GB volumes in Figure 5.21).

The second Unicage implementation of the *aggregation* workload was slower than the Spark implementation for the 347 GB and 698 GB data sets, but when the performance of Spark degraded due to the recomputing of RDD partitions, the Unicage and Spark implementations became closely matched in performance (compare the Spark and Unicage bars for the 1407 GB and larger volumes in Figures 5.20 and 5.21).

5.4 Discussion

The challenges faced during the development of the workload implementations and the reflection on the results obtained from the benchmarks prompted the following discussion.

Complex big data systems offer a plethora of abstractions to ease the implementation and execution of big data processing workloads. In our experimental setup, HDFS provided the abstraction of a distributed file system, and YARN provided distributed resource management to the Hadoop cluster. Unicage is characterized by an absence of these abstractions.

What were the felt impacts of using a distributed file system?

A sorted data set has inter-record dependencies, meaning each record has a placement dependency on other records. Unicage requires output data sets with this property to be stored in a single machine. For this reason, and because the *sort* workload preserves the volume of the data set, even though the *sort* implementation with Unicage failed prematurely, it was predicted to fail for data sets larger than 1024 GB, since that was the maximum storage capacity of the *unicageleader* node (see Table 4.4 in Chapter 4), where the outputs of the workloads were consolidated. In this aspect, Unicage can scale horizontally[2] to improve the performance of the workloads, but is hindered by the capacity of singular machines, being highly dependent on vertical scaling[3] to address larger volumes. This is limitative, as a machine can only scale

[2] **horizontal scaling**: adding more machines to the cluster to cope with demands.
[3] **vertical scaling**: adding more resources to existing machines in the cluster to cope with demands.

so much before it becomes impractical to maintain. HDFS allows large inter-record dependent outputs to be stored across multiple nodes. This is not immediately possible in lean big data systems without the integration of additional systems.

A similar comment can be made regarding the *join* workload, which explores the inter-record dependencies of the input data set. To join two records from different tables, Unicage required these to be stored in the same node. As such, the *join* implementation with Unicage required intermediate files to be moved around the cluster, severely crippling the performance. Having joinable records in different nodes was not a challenge for the tested complex big data systems, as applications can access HDFS and perform the operation directly over it.

As a consequence of the aforementioned challenges, lean implementations with shell scripts tend to be I/O-intensive. This has motivated recent research on the offloading of shell computation to where the accessed data is located, such is the proposition of POSH [19].

What were the felt impacts of using a distributed resource manager?

Another challenge of lean big data systems was the optimization of the resource usage, which was a manual endeavor that was only possible with the assistance of observability tools. This process was also error prone, as the resource usage patterns of shell script implementations often became difficult to predict with the increase of the input data set volumes. This was the case with the *aggregation* workload, where a refactoring was necessary after the first Unicage implementation failed for the 347 GB data set, due to the unpredictable memory usage of the `msort` command when met with the increased volume of the intermediate files.

Optimizing resource usage was not a challenge with the tested complex big data systems, as that is the main function of YARN. YARN interprets the configuration of properties (e.g., `spark.executor.instances` and `spark.executor.cores` in Spark) and manages resources accordingly, regardless of the workload or the volume of the input data sets.

Parallelizing shell commands is one of the techniques to optimize the usage of resources and the performance of lean implementations, as shown in Section 5.1. Recent research has addressed some of the burdens of this manual endeavor with solutions to automatically parallelize shell scripts, such is the proposition of PASH [18].

What was the performance impact of loading data sets with different file system layouts?

The Wikipedia text entries data sets were comprised of a growing number of 500 MB files, while the e-commerce tables data sets were comprised of a fixed number of 8 files with growing volumes. Table 5.1 highlights these differences for the largest tested volumes.

Data set	Real volume	Number of files	Volume per file
Wikipedia text entries	8019 GB	16380 files	0.490 GB
E-commerce *item* tables	5672 GB	4 files	1418 GB
E-commerce *order* tables	3745 GB	4 files	936 GB

Table 5.1: Number of files and estimated volume per file of the largest tested input data sets.

Figure 5.22 shows a composite chart of the loading rates of all benchmarked data loading workloads. The vertical axis shows the loading rate, in gigabytes per second, and the horizontal axis shows the tiered input data set volumes, in gigabytes. Like in the loading rate charts, higher rate bars mean better performance. For the structured e-commerce tables data sets, the process of creating the Hive tables in the Hadoop cluster was excluded from the chart, as that is a consequence of the structural properties of the data sets and not of their layouts in the file system. As far as HDFS is aware, data sets are just files.

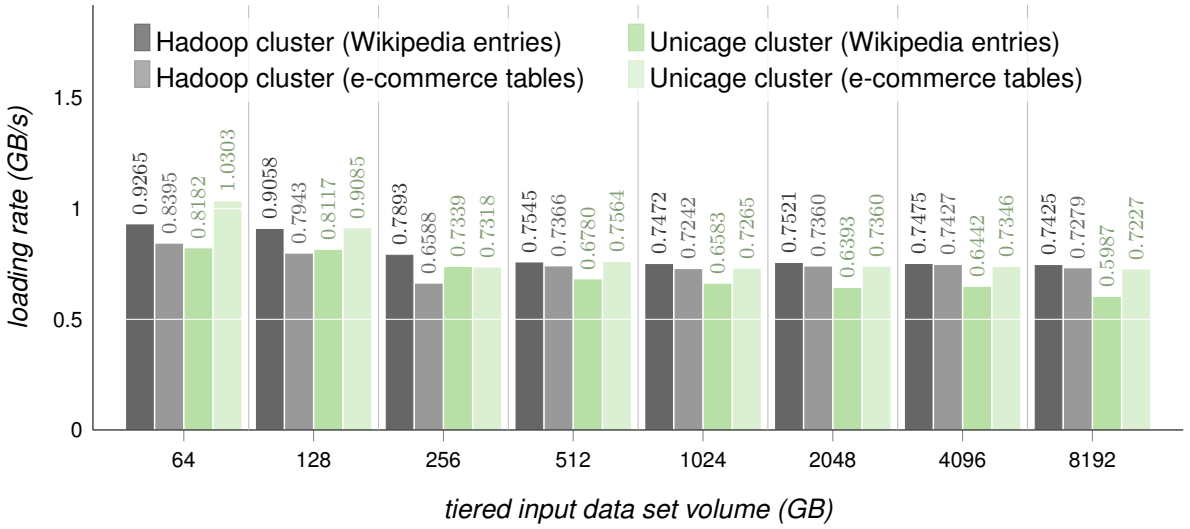


Figure 5.22: Data loading: composite chart of loading rates.

The performance of loading data into the Hadoop cluster with the `put` operation of HDFS is slightly increased for data sets with a large number of small files, providing each file is large enough to fill file blocks in the file system (128 MB each, by default), such was the case of the Wikipedia text entries data sets (compare the two Hadoop bars for each volume in Figure 5.22). The performance of loading data into the Unicage cluster with the `distr-distr` command is slightly increased for data sets with a small number of large files, such as the e-commerce tables data sets (compare the two Unicage bars for each volume in Figure 5.22).

5.5 Chapter Summary

This chapter covered the results obtained with the benchmarks, summarized as follows:

- Data loading with Unicage was slightly slower than with HDFS for unstructured data, but significantly faster for structured data, across all tested data set volumes.
- For the *grep* workload, the performance of Unicage was superior to that of the complex big data systems across all tested data set volumes.
- For the *sort* workload, the performance of Unicage was superior to that of the complex big data systems for volumes up to 251 GB, but Unicage failed to produce correct outputs for the 501 GB and larger data sets.
- For the *wordcount* workload, the performance of Unicage was inferior to that of Spark, but superior to that of Hadoop for volumes up to 4009 GB. Spark failed the *wordcount* workload for the 8019 GB data set, so Unicage was the only system capable of completing the workload within reasonable time.
- For the *select* workload, the performance of Unicage was superior to that of the complex big data systems across all tested data set volumes.
- For the *join* workload, the performance of Unicage was significantly inferior to that of the complex big data systems for the 68 GB and 139 GB data sets. Unicage failed to produce correct outputs for the 281 GB and larger data sets.
- For the *aggregation* workload, the performance of Unicage was superior to that of the complex big data systems for volumes up to 173 GB. Unpredictable resource usage patterns required the Unicage implementation to be refactored at the cost of performance, which was inferior to that of Spark but superior to that of Hive for the 347 GB and 698 GB data sets. The performance of Unicage was closely matched with that of Spark and superior to that of Hive for the 1407 GB and larger data sets.

We have also addressed how the absence of an abstraction similar to that of a distributed file system is a limitative factor for lean big data systems, especially with inter-record dependent workloads, such as the *sort* and *join* workloads. The absence of a distributed resource management abstraction is also limitative for lean big data systems, as the resource usage patterns of shell scripts can become increasingly unpredictable with the increase of the input data set volumes.

Chapter 6

Conclusion

This chapter consolidates the conclusions of this study. Section 6.1 provides answers to the research questions we set out to answer. Section 6.2 highlights the achievements of the study. Section 6.3 revisits some aspects we would have done differently, in hindsight. Section 6.4 provides suggestions for future work. Section 6.5 closes the study with some final words.

6.1 Answering the Research Questions

We benchmarked the performance of batch processing workloads over unstructured data sets and query processing workloads over structured data sets. We also benchmarked the data loading performance. The tiered volumes of the data sets ranged from 64 GB to 8192 GB.

In the following answers to the research questions, the data sets with tiered volumes ranging from 64 GB to 256 GB are referred to as *small*; those from 512 GB to 4096 GB are referred to as *medium*; and those with 8192 GB are referred to as *large*.

RQ 1: ***In a cloud computing cluster setting, how do complex and lean big data systems compare from a performance perspective?***

We have shown that, from a performance perspective, there are big data workloads that do not require the use of complex big data systems.

For search workloads (*grep* and *select*), Unicage shell scripts were preferred to Spark across all tested input data set volumes; for grouping workloads (*wordcount* and *aggregation*), Unicage was only preferred to Spark for large data sets; for the *sort* workload, Unicage was only preferred to Spark for small data sets; and for the *join* workload, Spark was preferred to Unicage across all tested input data set volumes. Figure 6.1 presents a flowchart that summarizes these findings, as they apply in the tested experimental environment.

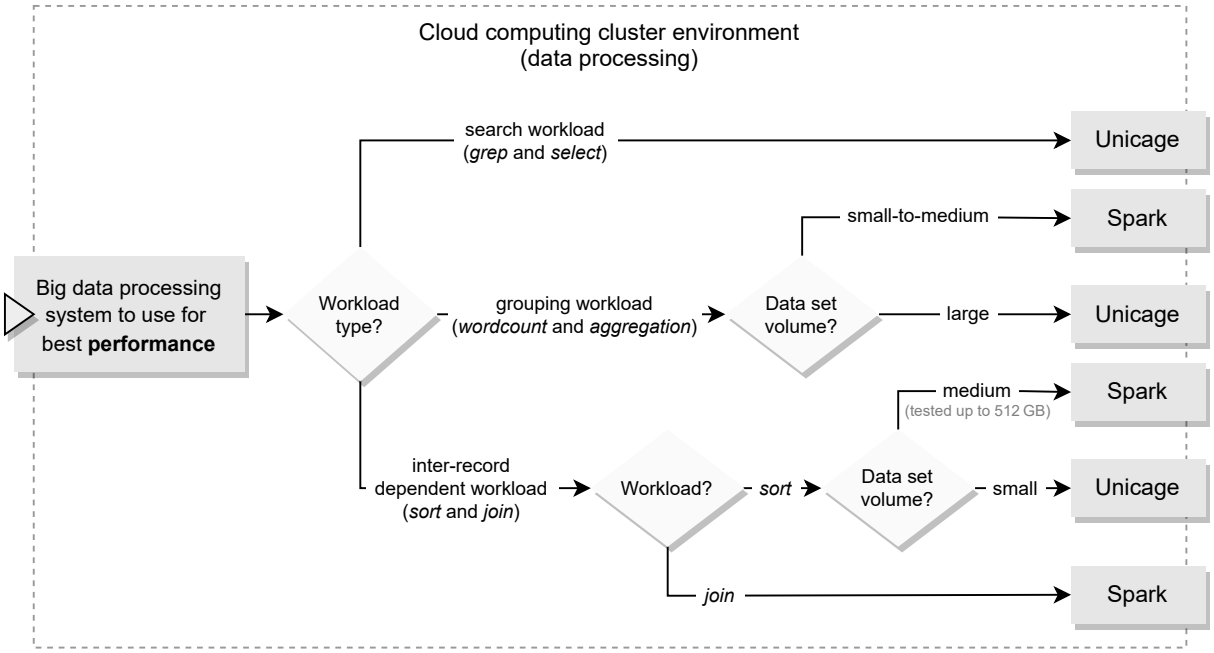


Figure 6.1: Summary of conclusions from the data processing benchmarks.

Regarding data loading, the results show that the performance of loading data sets into a Unicage cluster is slightly inferior for unstructured data and significantly superior for structured data, than that of loading the same data sets into a Hadoop cluster with HDFS and Hive.

The following answers to the remaining research questions provide deeper context to the previous statements.

RQ 2: ***In a cloud computing cluster setting, how do complex and lean big data systems handle the volume of big data?***

Lean implementations force the developer to think critically about the volume of the target input data sets, whose increase is often accompanied by unpredictable resource usage patterns. Optimizing lean implementations is a manual endeavor and is error-prone, thus increases the probability of failed execution for small-to-medium data sets. This is not a problem for big data systems that rely on distributed file systems and distributed resource managers to deal with very large data set volumes, provided the underlying cluster architecture supports them.

Looking at the volumes for which each big data system failed to produce correct outputs for corroborates the previous statement. Table 6.1 shows an inventory of all completed benchmarks. It highlights the successfully completed workloads (✔), and the workloads the tested big data systems failed to produce correct outputs for (✘).

		Tiered volumes (GB)							
		—— Small ——			———— Medium ————				— Large —
Workload	Big data system	64	128	256	512	1024	2048	4096	8192
Grep	Hadoop	✔	✔	✔	✔	✔	✔	✔	✔
	Spark	✔	✔	✔	✔	✔	✔	✔	✔
	Unicage	✔	✔	✔	✔	✔	✔	✔	✔
Sort	Hadoop	✔	✔	✔	✔[a]	n/a[b]	n/a[b]	n/a[b]	n/a[b]
	Spark	✔	✔	✔	✔	n/a[b]	n/a[b]	n/a[b]	n/a[b]
	Unicage	✔	✔	✔	✘				
Wordcount	Hadoop	✔	✔	✔	✔	✔	✔	✔	✔[a]
	Spark	✔	✔	✔	✔	✔	✔	✔	✘
	Unicage	✔	✔	✔	✔	✔	✔	✔	✔
Select	Hive	✔	✔	✔	✔	✔	✔	✔	✔
	Spark	✔	✔	✔	✔	✔	✔	✔	✔
	Unicage	✔	✔	✔	✔	✔	✔	✔	✔
Join	Hive	✔	✔	✔	✔	✔	✔	✔	✔
	Spark	✔	✔	✔	✔	✔	✔	✔	✔
	Unicage	✔	✔	✘					
Aggregation	Hive	✔	✔	✔	✔	✔	✔	✔	✘
	Spark	✔	✔	✔	✔	✔	✔	✔	✔
	Unicage	✔	✔	✔	✔[c]	✔[c]	✔[c]	✔[c]	✔[c]

[a] Unreported single-runs, with no statistical validity, used to verify the output of another big data system.

[b] Benchmarks that were not performed due to a lack of an efficient verification method.

[c] Benchmarks that required a different implementation to proceed, due to unpredictable resource usage patterns.

Table 6.1: List of completed benchmarks.

Hadoop, Hive and Spark were able to finish all workloads, with the exception of Spark and Hive failing the *wordcount* and *aggregation* workloads, respectively, for the largest tested volumes. The implementations in these systems were straightforward, as they rely on HDFS for distributed file system operations and on YARN for distributed resource management. The absence of these abstractions was felt during the development of lean implementations, as it forced the management of the resource usage of the shell scripts to be a manual endeavor, and was a culprit in the premature failure of the *sort*, *join* and *aggregation* workloads for small-to-medium volumes.

We have seen that Unicage requires inter-record dependent outputs to be consolidated in a single machine. The *sort* workload is one such example. In these cases, Unicage is dependent on the vertical scaling of that machine to address larger data sets, which becomes impractical with the increase of the volumes. A distributed file system like HDFS eliminates this problem by allowing such outputs to be stored across multiple nodes, being solely dependent on horizontal scaling to meet storage needs.

RQ 3: ***In a cloud computing cluster setting, how do complex and lean big data systems handle the variety of big data?***

With complex big data systems, the type of the target data sets dictates what subsystems should be installed in the software stack. Hadoop MapReduce is used mostly for unstructured data. Hive can only be used for structured data. Spark can be used for most varieties of data, including streaming data, but requires an underlying data warehousing solution to house structured data, such as the Metastore of Hive.

As for lean big data systems, any variety of data is supported as long as it can be represented as text and processed with a shell script-based implementation. Recognizing the type of data to be processed, and finding the most efficient way to parse and convert that data into text files with a normalized format are the recurrent first steps of writing lean implementations with Unicage. This is a human endeavor, and the developer needs to be well aware of the structural properties of the data to ensure its veracity is preserved.

Data loading is also a divisive factor between complex and lean big data systems when it comes to the variety of the data sets. For unstructured data, the performance of loading a data set with Unicage is slightly worse than loading the same data set into HDFS. For structured data, Unicage does not require the knowledge of the schema directly, so it does not incur in the additional time required by complex systems to create the tables in the underlying database, and thus performs significantly better. The trade-off is once again that, in the absence of formal schema, it is of the responsibility of the developer to ensure the veracity of the data is preserved.

A consequence of the previous statements is that if a data set is of the structured type and has an elaborate schema, it is not ideal to delegate the responsibility of maintaining its structure to the developer, as that might compromise the veracity of the data. In this context, complex systems with abstractions to formalize the structure of the data set (e.g., the Metastore of Hive) should be used.

Figure 6.2 presents a flowchart that summarizes the main points of the previous statements. It highlights the varieties of data for which Unicage *can* be used; those for which Unicage is not

ideal, and thus a complex system *should* be used; and those for which Unicage cannot be used, and thus a complex system *must* be used.

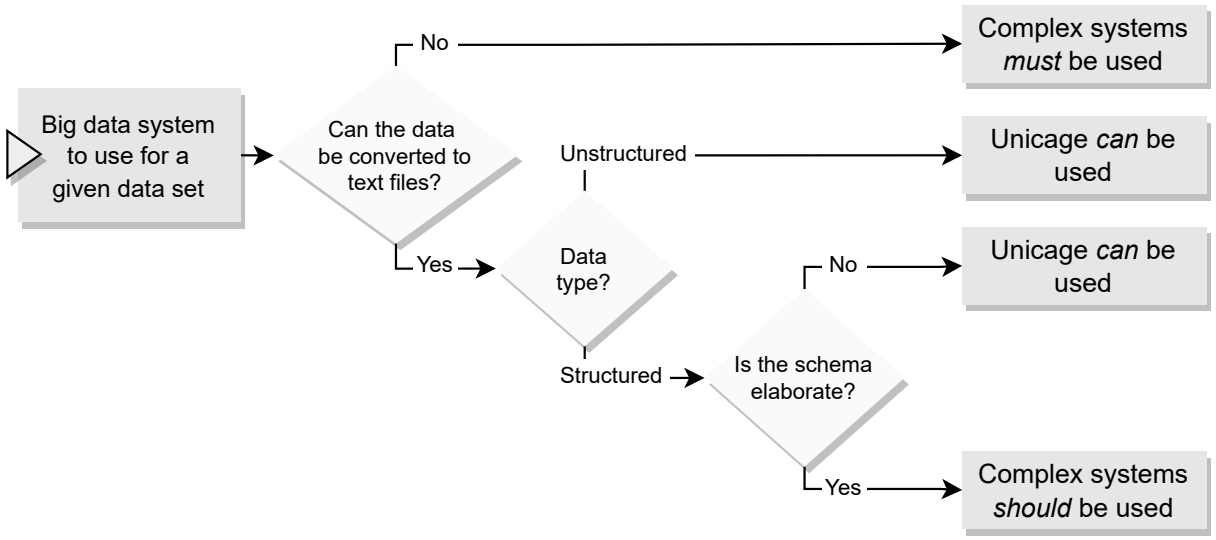


Figure 6.2: Summary of the varieties of data Unicage can be used with.

In complex systems, the variety of big data is supported by the selectively installed subsystems of the software stack, while in lean systems, it is supported by the knowledge of the developer and implemented algorithms for the workloads.

6.2 Achievements

This study resulted in two major achievements. The first comprises the results and conclusions, as comprehensively compiled in this document. The second is the big data benchmark suite, purposefully implemented with the novelty of considering both complex and lean big data systems. This suite is a valuable asset for future research, as it has the following features:

- Extensive documentation on cloud provisioning using the VPC infrastructure of IBM Cloud, on how to use the big data benchmark suite, and on how to reproduce and extend the benchmarks;

- Shell scripts for automated configuration of the experimental clusters with the deployment and localization of the various required tools, data generators (BDGS and PDGF), big data systems and libraries (Apache Hadoop, Hive, Spark, Unicage Tukubai and BOA), and observability tools for metric collection (Netdata);

- Shell script implementations for the generation and loading of unstructured and structured data sets (Wikipedia text entries and e-commerce tables, respectively) into the experimental clusters;
- Implementations of six big data processing workloads (*grep*, *sort*, *wordcount*, *select*, *join* and *aggregation*) in Apache Hadoop, Hive, Spark, and using shell script-based Unicage solutions;
- Shell scripts wrapping the automated preparation of the environments, execution of the workloads, profiling of the metrics, and verification of the results of all benchmarks addressed in this study.

6.3 Retrospective

In hindsight, there are some aspects of this study we wish we could have addressed differently. The impacts range from a better management of the allocated time for the study, to the improved reproducibility of the benchmarks.

A single Hadoop cluster was used for all complex big data systems. This lack of isolation meant a Hadoop or Hive workload and a Spark workload could not be simultaneously benchmarked. With a second identical instance of the Hadoop cluster, we could have done these benchmarks simultaneously and saved time.

The data generators used uncontrollable input seeds, either resulting from the starting timestamps or pseudo-random values. The reproducibility of the benchmarks could have been improved had we used controllable input seeds for the generation of the input data sets. The volumes produced by the data generators also had some deviance when compared with the planned volumes. This could have been minimized had we invested more time calibrating the data generators.

The *select* and *aggregation* workloads used only one of the two tables of the structured data sets used for all query processing benchmarks, with a fraction of the total volumes. The resulting conclusions could have been richer if the data sets were generated with a single table that met the desired volumes, exclusively for the *select* and *aggregation* benchmarks.

6.4 Future Work

Our findings favor further work on complex and lean big data systems. Future studies should address more complex sets of workloads in application-specific scenarios. Other types of data

processing should also be benchmarked, including stream processing and machine learning. Benchmarking stream processing workloads is noteworthy, as it can provide conclusions regarding how each system handles the *V property* we did not address: the *velocity* of big data.

The obstacles found in the *sort* and *join* workloads motivate the future integration of distributed file system and resource usage abstractions with lean big data systems. The integration of recent technologies like PaSh [18] and Posh [19] can also provide further performance optimizations to lean shell script-based implementations.

We have seen that big data systems are characterized by a set of quality attributes (e.g., performance, scalability, fault tolerance, security and usability). Our findinds focus strictly on the performance, with comments regarding the scalability and usability of these systems. Future studies should compare complex and lean big data systems with other quality attributes in mind, such as fault tolerance and security.

6.5 Final Words

Complex big data systems are widely used and provide abstractions that lean big data systems do not. The decision between taking the complex or lean approaches will always depend on the use-case, with multiple weighting factors, including the types of workloads, the volumes and varieties of the data sets, and the available resources in the computing clusters. However, our study provides evidence that lean implementations are enough to achieve superior or comparable performance for some big data processing workloads with terabyte-sized data sets. Our results vouch for the possibility of combining the two approaches to obtain the best performance and the better use of resources in a cloud environment. For instance, lean implementations can be used to do simple pre-processing tasks (e.g., search workloads) before data is retained and processed in complex big data systems, saving resources and consequently reducing costs. This can be especially relevant if combined with edge computing, as it is expected to support many Internet of Things applications, where the edge nodes with limited computational and power resources will be able to run lean software stacks but not complex ones.

Bibliography

[1] M. Rothmuller and S. Barker. IoT - The internet of transformation 2020. Technical report, Juniper Research, 2020.

[2] A. Holst. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. Technical report, Statista, 2021.

[3] J. Wang, Y. Yang, T. Wang, R. S. Sherratt, and J. Zhang. Big data service architecture: A survey. *Journal of Internet Technology*, 21(2):393–405, 2020.

[4] A. De Mauro, M. Greco, and M. Grimaldi. A formal definition of big data based on its essential features. *Library Review*, 2016.

[5] I. O. Pappas, P. Mikalef, M. N. Giannakos, J. Krogstie, and G. Lekakos. Big data and business analytics ecosystems: Paving the way towards digital transformation and sustainable societies. *Information Systems and e-Business Management*, 16(3):479–491, 2018.

[6] E. B. Allen, T. M. Khoshgoftaar, and Y. Chen. Measuring coupling and cohesion of software modules: An information-theory approach. In *Proceedings Seventh International Software Metrics Symposium*, pages 124–134. IEEE, 2001.

[7] A. Trivedi, P. Stuedi, J. Pfefferle, R. Stoica, B. Metzler, I. Koltsidas, and N. Ioannou. On the [ir] relevance of network performance for data processing. In *8th USENIX Workshop on Hot Topics in Cloud Computing*, 2016.

[8] J. M. Moreira, H. Galhardas, and M. L. Pardal. LeanBench: Comparing software stacks for batch and query processing of IoT data. *Procedia Computer Science*, 130:448–455, 2018.

[9] C. Batini, E. Nardelli, and R. Tamassia. A layout algorithm for data flow diagrams. *IEEE Transactions on Software Engineering*, (4):538–546, 1986.

[10] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. BigDataBench: A big data benchmark suite from internet services. In *20th IEEE*

International Symposium on High Performance Computer Architecture, pages 488–499. IEEE, 2014.

[11] W. Gao, J. Zhan, L. Wang, C. Luo, D. Zheng, X. Wen, R. Ren, C. Zheng, X. He, H. Ye, et al. BigBataBench: A scalable and unified big data and AI benchmark suite. *arXiv preprint arXiv:1802.08254*, 2018.

[12] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Conference*, pages 119–130, 1985.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[14] D. S. Dean Hildebrand. Colossus under the hood: A peek into Google's scalable storage system, 2021. URL `https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system`. (Visited on 09/22/2022).

[15] K. V. Shvachko, H. Kuang, S. R. Radia, and R. J. Chansler. The Hadoop Distributed File System. *26th IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–10, 2010.

[16] M. Gancarz. *Linux and the Unix philosophy*. Digital Press, 2003. ISBN 9781555582739.

[17] W. Shotts. *The Linux command line*. No Starch Press, 2009. ISBN 9781593274269.

[18] N. Vasilakis, K. Kallas, K. Mamouras, A. Benetopoulos, and L. Cvetković. PASH: Light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 49–66, 2021.

[19] D. Raghavan, S. Fouladi, P. Levis, and M. Zaharia. POSH: A data-aware shell. In *USENIX Annual Technical Conference*, pages 617–631, 2020.

[20] S. Handa, K. Kallas, N. Vasilakis, and M. C. Rinard. An order-aware dataflow model for parallel Unix pipelines. *Proceedings of the ACM on Programming Languages*, 5:1–28, 2021.

[21] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[22] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.

[24] B. Dong, Q. Zheng, F. Tian, K.-M. Chao, R. Ma, and R. Anane. An optimized approach for storing and accessing small files on cloud storage. *Journal of Network and Computer Applications*, 35(6):1847–1862, 2012.

[25] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a Map-Reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - A petabyte scale data warehouse using Hadoop. In *26th IEEE International Conference on Data Engineering*, pages 996–1005. IEEE, 2010.

[27] E. Costa, C. Costa, and M. Y. Santos. Evaluating partitioning and bucketing strategies for hive-based big data warehousing systems. *Journal of Big Data*, 6(1):34, 2019.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[29] A. Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.

[30] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation*, pages 15–28, 2012.

[31] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1383–1394, 2015.

[32] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 423–438, 2013.

[33] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in Apache Spark. In *Proceedings of the International Conference on Management of Data*, pages 601–613, 2018.

[34] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 599–613, 2014.

[35] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. MLlib: Machine learning in Apache Spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[36] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, 2010.

[37] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of Map-Reduce: The Pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[38] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, 2008.

[39] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[40] Sally. Apache in 2020 - By the digits, 2021. URL `https://news.apache.org/foundation/entry/apache-in-2020-by-the`. (Visited on 09/27/2022).

[41] Sally. Apache in 2021 - By the digits, 2022. URL `https://news.apache.org/foundation/entry/apache-in-2021-by-the`. (Visited on 09/27/2022).

[42] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 256–267, 2017.

[43] K. Loudon. *Mastering algorithms with C*. O'Reilly Media, Inc., 1999. ISBN 9781565924536.

[44] R. Han, L. K. John, and J. Zhan. Benchmarking big data systems: A review. *IEEE Transactions on Services Computing*, 11(3):580–597, 2017.

[45] W. Gao, J. Zhan, L. Wang, C. Luo, D. Zheng, F. Tang, B. Xie, C. Zheng, X. Wen, X. He, et al. Data motifs: A lens towards fully understanding big data and AI workloads. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–14, 2018.

[46] Z. Ming, C. Luo, W. Gao, R. Han, Q. Yang, L. Wang, and J. Zhan. BDGS: A scalable big data generator suite in big data benchmarking. In *Advancing Big Data Benchmarks*, pages 138–154. Springer, 2013.

[47] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A data generator for cloud-scale benchmarking. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 41–56. Springer, 2010.

[48] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. BigBench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1197–1208, 2013.

[49] A. Ghazal, T. Ivanov, P. Kostamaa, A. Crolotte, R. Voong, M. Al-Kateb, W. Ghazal, and R. V. Zicari. BigBench v2: The new and improved BigBench. In *33rd IEEE International Conference on Data Engineering*, pages 1225–1236. IEEE, 2017.

[50] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *26th IEEE International Conference on Data Engineering Workshops*, pages 41–51. IEEE, 2010.

[51] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of data*, pages 165–178, 2009.

[52] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. SparkBench: A comprehensive benchmarking suite for in memory data analytic platform Spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, pages 1–8, 2015.

[53] M. Ferreira, A. Neves, R. Gorjao, C. Cruz, and M. L. Pardal. Smart meter data processing: A showcase for simple and efficient textual processing. *[Private communication]*, 2021.

[54] R. Rivest. The MD5 message-digest algorithm. Technical report, 1992.

[55] D. C. Montgomery and G. C. Runger. *Applied statistics and probability for engineers*. John Wiley & Sons, 2010.

Appendix A

Statistical Validation

An experiment aims to reach conclusions regarding how a given *population* behaves in relation to a given *variable*. Typically, an experiment does not attempt to survey the whole of a population, as it is in most cases impossible. Instead, a subset of the population is surveyed. This subset is called a *sample*. One cannot immediately assume an observed value for a variable is representative of the whole population as much as it is representative of the surveyed sample. Therefore, the statistical validation process is imperative to estimate how the behavior of the population may deviate from the behavior of the sample. Section A.1 provides the necessary background, inspired by Montgomery et al. [55], to statistically validate the results collected throughout this study. Section A.2 provides that validation.

A.1 Population Mean Estimation and Confidence Interval

In an experiment that surveys a sample of a population regarding variable *VariableA*, in seconds, the observed value for said variable is a representative of the surveyed sample, but it is not a representative of the population. To estimate the behavior of the population in relation to *VariableA*, one must establish a confidence level *p* and estimate the population mean value μ and confidence interval *CI*:

$$\textit{VariableA} = \mu \pm \textit{CI} \text{ seconds, with } p \text{ of confidence.}$$

This means there is a *p* probability of the mean distribution of *VariableA* being in the interval $[\mu - \textit{CI}, \mu + \textit{CI}]$. There are three fundamental steps to perform this estimation.

Step A: Number of samples *n*

The first step is to perform *n* observations of the experiment. The number of samples *n* directly influences the confidence interval *CI* of *VariableA*. More samples mean a smaller confidence interval, which directly translates to a smaller margin of error *ME*. The margin of error can also be lessened by admitting a smaller confidence level *p*.

Step B: Sample mean value $\bar{x}$ and standard deviation *s*

Let x_i be the observed value of *VariableA* in the *i*-th sample of the experiment. The second step is to calculate, from the obtained results, the sample mean value $\bar{x}$ and the sample standard deviation *s*:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{A.1}$$

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \tag{A.2}$$

Step C: Estimated population mean value μ and confidence interval *CI*

The third step is to choose a confidence level *p* and to calculate, from $\bar{x}$ and *s*, the estimated population mean value μ and the confidence interval *CI*. The mean sample value $\bar{x}$ is considered a valid estimation of the mean population value μ if and only if the following conditions are met: i) the value of any sample is not dependent on the values of the other samples; ii) all samples come from the same population with estimated population mean value μ and population standard deviation σ. Consequently, under these conditions, the sample standard deviation *s* is considered a valid estimation of the population standard deviation σ. When these conditions are met, the *Central Limit Theorem* (CLT) can be applied to calculate the confidence interval *CI* around μ.

The CLT states that when the number of samples is sufficiently large (typically $n \geq 30$), the sample mean value $\bar{x}$ has a standard normal distribution. When the number of samples is smaller ($n < 30$), the sample mean value $\bar{x}$ has a Student's *t*-distribution. The latter is the case with all experiments performed throughout this study, so for $n < 30$, the confidence interval *CI* is calculated as follows:

$$CI = t_{\alpha,df}\frac{s}{\sqrt{n}} \tag{A.3}$$

The significance level α is given by $\alpha = (1 - p) \div 2$, and the degrees of freedom *df* are given by $df = n - 1$. These are the coordinates that allow the value of $t_{\alpha,df}$ to be found in the precomputed *t*-distribution table. Montgomery et al. [55] provides this table, and an excerpt is provided in Table A.1.

					α					
df	0.4	0.25	0.1	0.05	0.025	0.01	0.005	0.0025	0.001	0.0005
1	0.325	1.000	3.078	6.314	12.706	31.821	63.657	127.321	318.309	636.619
2	0.289	0.816	1.886	2.920	4.303	6.965	9.925	14.089	22.327	31.599
3	0.277	0.765	1.638	2.353	3.182	4.541	5.841	7.453	10.215	12.924
4	0.271	0.741	1.533	2.132	2.776	3.747	4.604	5.598	7.173	8.610
5	0.267	0.727	1.476	2.015	2.571	3.365	4.032	4.773	5.893	6.869
					...					

Table A.1: Student's t-distribution table.

With the estimations of the population mean value μ and confidence interval *CI*, the margin of error *ME* in % is calculated as follows:

$$ME = \frac{CI}{\mu} \times 100 \tag{A.4}$$

A.2 Validating the Performed Experiments

The statistical validation of the results collected for the execution time metric when running the *wordcount* workload with Spark on a 63 GB input data set is used as a practical example of the application of the technique presented in Section A.1. The benchmark was run a total of three times. The first run took 214 seconds to complete. The second run took 201 seconds. The third run took 199 seconds. Let the chosen confidence level *p* be 95%:

$$\begin{aligned} p &= 95\% \text{ confidence} \\ n &= 3 \text{ samples} \\ \bar{x} &= 204.67 \text{ seconds} \\ s &= 8.14 \text{ seconds} \end{aligned}$$

$$
\begin{aligned}
\text{Execution time} &= 204.67 \pm t_{(1-0.95)\div 2,3-1}\frac{8.14}{\sqrt{3}} \text{ seconds} \\
&= 204.67 \pm t_{0.025,2}\frac{8.14}{\sqrt{3}} \text{ seconds} \\
&= 204.67 \pm 4.303\frac{8.14}{\sqrt{3}} \text{ seconds} \\
&= 204.67 \pm 20.22 \text{ seconds with 95\% confidence}
\end{aligned}
$$

This result means that we are 95% confident that if the benchmark was repeated a very large amount of times, approaching what we would consider a population, the mean distribution of the execution time would be within the interval [184.45, 224.89], which translates to a margin of error *ME* of 9,88%.

Tables A.2, A.3, A.4, A.5, A.6, A.7, A.8 and A.9 compile the statistically validated results of all benchmarks performed throughout this study.

Input data set volume	Loading process	Mean sample execution time	Confidence levels		
			90%	95%	99%
63 GB	Loading with HDFS	67.67 seconds	± 6.36 seconds	± 9.41 seconds	± 21.72 seconds
	Loading with Unicage	77.33 seconds	± 8.31 seconds	± 12.25 seconds	± 28.25 seconds
125 GB	Loading with HDFS	138.33 seconds	± 8.65 seconds	± 12.74 seconds	± 29.40 seconds
	Loading with Unicage	154.00 seconds	± 11.68 seconds	± 17.22 seconds	± 39.71 seconds
250 GB	Loading with HDFS	317.67 seconds	± 4.87 seconds	± 7.18 seconds	± 16.56 seconds
	Loading with Unicage	342.00 seconds	± 9.39 seconds	± 13.84 seconds	± 31.92 seconds
501 GB	Loading with HDFS	663.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
	Loading with Unicage	739.33 seconds	± 5.92 seconds	± 8.72 seconds	± 20.11 seconds
1002 GB	Loading with HDFS	1341.33 seconds	± 2.58 seconds	± 3.80 seconds	± 8.77 seconds
	Loading with Unicage	1521.67 seconds	± 5.16 seconds	± 7.60 seconds	± 17.53 seconds
2005 GB	Loading with HDFS	2665.67 seconds	± 10.30 seconds	± 15.18 seconds	± 35.01 seconds
	Loading with Unicage	3136.00 seconds	± 163.53 seconds	± 240.96 seconds	± 555.82 seconds
4009 GB	Loading with HDFS	5363.00 seconds	± 16.25 seconds	± 23.95 seconds	± 55.24 seconds
	Loading with Unicage	6223.00 seconds	± 71.78 seconds	± 105.77 seconds	± 243.99 seconds
8019 GB	Loading with HDFS	10800.00 seconds	± 181.08 seconds	± 266.82 seconds	± 615.47 seconds
	Loading with Unicage	13393.33 seconds	± 2117.40 seconds	± 3120.03 seconds	± 7196.90 seconds

Table A.2: Data loading - Wikipedia text entries unstructured data set: statistically validated execution times.

			Confidence levels		
Input data set volume	Big data system	Mean sample execution time	90%	95%	99%
63 GB	Hadoop	211.00 seconds	± 1.69 seconds	± 2.48 seconds	± 5.73 seconds
	Spark	112.00 seconds	± 1.69 seconds	± 2.48 seconds	± 5.73 seconds
	Unicage	69.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
125 GB	Hadoop	392.00 seconds	± 6.09 seconds	± 8.97 seconds	± 20.69 seconds
	Spark	190.33 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
	Unicage	136.33 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
250 GB	Hadoop	774.00 seconds	± 2.92 seconds	± 4.30 seconds	± 9.91 seconds
	Spark	368.67 seconds	± 20.94 seconds	± 30.85 seconds	± 71.17 seconds
	Unicage	270.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
501 GB	Hadoop	1505.67 seconds	± 20.25 seconds	± 29.83 seconds	± 68.82 seconds
	Spark	690.00 seconds	± 25.51 seconds	± 37.59 seconds	± 86.70 seconds
	Unicage	538.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
1002 GB	Hadoop	2969.33 seconds	± 2.58 seconds	± 3.80 seconds	± 8.77 seconds
	Spark	1358.67 seconds	± 23.62 seconds	± 34.80 seconds	± 80.28 seconds
	Unicage	1074.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
2005 GB	Hadoop	5895.33 seconds	± 15.66 seconds	± 23.08 seconds	± 53.23 seconds
	Spark	2671.33 seconds	± 87.14 seconds	± 128.41 seconds	± 296.19 seconds
	Unicage	2146.33 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
4009 GB	Hadoop	11884.33 seconds	± 83.01 seconds	± 122.32 seconds	± 282.15 seconds
	Spark	5378.67 seconds	± 184.55 seconds	± 271.94 seconds	± 627.28 seconds
	Unicage	4290.33 seconds	± 3.89 seconds	± 5.74 seconds	± 13.24 seconds
8019 GB	Hadoop	24057.67 seconds	± 86.00 seconds	± 126.72 seconds	± 292.29 seconds
	Spark	18711.00 seconds	± 590.17 seconds	± 869.62 seconds	± 2005.94 seconds
	Unicage	8580.67 seconds	± 15.11 seconds	± 22.26 seconds	± 51.34 seconds

Table A.3: Data processing - *grep*: statistically validated execution times.

Input data set volume	Big bata system	Mean sample execution time	Confidence levels		
			90%	95%	99%
63 GB	Hadoop	13043.67 seconds	± 67.23 seconds	± 99.07 seconds	± 228.52 seconds
	Spark	2756.00 seconds	± 8.92 seconds	± 13.14 seconds	± 30.31 seconds
	Unicage	1551.67 seconds	± 16.37 seconds	± 24.12 seconds	± 55.64 seconds
125 GB	Hadoop	27893.67 seconds	± 224.18 seconds	± 330.34 seconds	± 761.99 seconds
	Spark	5088.33 seconds	± 11.83 seconds	± 17.44 seconds	± 40.23 seconds
	Unicage	3207.67 seconds	± 101.29 seconds	± 149.25 seconds	± 344.27 seconds
250 GB	Hadoop	62577.67 seconds	± 816.43 seconds	± 1203.02 seconds	± 2774.98 seconds
	Spark	10100.67 seconds	± 82.84 seconds	± 122.07 seconds	± 281.58 seconds
	Unicage	6446.00 seconds	± 24.95 seconds	± 36.77 seconds	± 84.81 seconds
501 GB	Hadoop	134774.00 seconds	No statistical validation (single run)		
	Spark	20147.33 seconds	± 305.46 seconds	± 450.10 seconds	± 1038.24 seconds
	Unicage		Incorrect output		
1002 GB	Hadoop		Not benchmarked		
	Spark		Not benchmarked		
	Unicage		Not benchmarked		
2005 GB	Hadoop		Not benchmarked		
	Spark		Not benchmarked		
	Unicage		Not benchmarked		
4009 GB	Hadoop		Not benchmarked		
	Spark		Not benchmarked		
	Unicage		Not benchmarked		
8019 GB	Hadoop		Not benchmarked		
	Spark		Not benchmarked		
	Unicage		Not benchmarked		

Table A.4: Data processing - *sort*: statistically validated execution times.

Input data set volume	Big data system	Mean sample execution time	Confidence levels 90%	95%	99%
63 GB	Hadoop	1081.00 seconds	± 19.00 seconds	± 28.00 seconds	± 64.58 seconds
	Spark	204.67 seconds	± 13.72 seconds	± 20.22 seconds	± 46.64 seconds
	Unicage	314.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
125 GB	Hadoop	2156.67 seconds	± 20.11 seconds	± 29.64 seconds	± 68.36 seconds
	Spark	370.67 seconds	± 8.65 seconds	± 12.74 seconds	± 29.40 seconds
	Unicage	636.00 seconds	± 23.42 seconds	± 34.50 seconds	± 79.59 seconds
250 GB	Hadoop	4265.67 seconds	± 73.60 seconds	± 108.46 seconds	± 250.18 seconds
	Spark	724.00 seconds	± 17.11 seconds	± 25.21 seconds	± 58.16 seconds
	Unicage	1274.00 seconds	± 1.69 seconds	± 2.48 seconds	± 5.73 seconds
501 GB	Hadoop	8480.67 seconds	± 196.65 seconds	± 289.77 seconds	± 668.42 seconds
	Spark	1585.33 seconds	± 228.84 seconds	± 337.20 seconds	± 777.81 seconds
	Unicage	2531.00 seconds	± 1.69 seconds	± 2.48 seconds	± 5.73 seconds
1002 GB	Hadoop	17022.67 seconds	± 109.48 seconds	± 161.32 seconds	± 372.11 seconds
	Spark	2987.67 seconds	± 48.16 seconds	± 70.97 seconds	± 163.71 seconds
	Unicage	5066.33 seconds	± 6.81 seconds	± 10.04 seconds	± 23.15 seconds
2005 GB	Hadoop	33611.00 seconds	± 45.27 seconds	± 66.70 seconds	± 153.85 seconds
	Spark	6724.67 seconds	± 8.48 seconds	± 12.50 seconds	± 28.82 seconds
	Unicage	10308.67 seconds	± 218.00 seconds	± 321.22 seconds	± 740.96 seconds
4009 GB	Hadoop	68032.67 seconds	± 1476.34 seconds	± 2175.41 seconds	± 5017.97 seconds
	Spark	14780.67 seconds	± 755.38 seconds	± 1113.07 seconds	± 2567.49 seconds
	Unicage	20643.00 seconds	± 72.14 seconds	± 106.30 seconds	± 245.19 seconds
8019 GB	Hadoop	135883.00 seconds	No statistical validation (single run)		
	Spark		Unable to finish (out-of-memory)		
	Unicage	41408.67 seconds	± 65.50 seconds	± 96.51 seconds	± 222.61 seconds

Table A.5: Data processing - *wordcount*: statistically validated execution times.

Input data set volume	Loading process	Mean sample execution time	Confidence levels		
			90%	95%	99%
68 GB	Loading with HDFS	81.33 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
	Creating Hive tables	315.33 seconds	± 5.92 seconds	± 8.72 seconds	± 20.11 seconds
	Loading with Unicage	65.67 seconds	± 3.51 seconds	± 5.17 seconds	± 11.92 seconds
139 GB	Loading with HDFS	174.67 seconds	± 5.16 seconds	± 7.60 seconds	± 17.53 seconds
	Creating Hive tables	562.00 seconds	± 7.35 seconds	± 10.83 seconds	± 24.98 seconds
	Loading with Unicage	152.67 seconds	± 19.47 seconds	± 28.69 seconds	± 66.18 seconds
281 GB	Loading with HDFS	380.60 seconds	± 3.51 seconds	± 5.17 seconds	± 11.92 seconds
	Creating Hive tables	1086.00 seconds	± 13.81 seconds	± 20.35 seconds	± 46.93 seconds
	Loading with Unicage	342.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
565 GB	Loading with HDFS	766.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
	Creating Hive tables	2122.33 seconds	± 51.69 seconds	± 76.16 seconds	± 175.69 seconds
	Loading with Unicage	746.67 seconds	± 4.25 seconds	± 6.26 seconds	± 14.44 seconds
1145 GB	Loading with HDFS	1581.33 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
	Creating Hive tables	4264.67 seconds	± 49.33 seconds	± 72.69 seconds	± 167.66 seconds
	Loading with Unicage	1576.33 seconds	± 2.58 seconds	± 3.80 seconds	± 8.77 seconds
2329 GB	Loading with HDFS	3152.00 seconds	± 3.37 seconds	± 4.97 seconds	± 11.46 seconds
	Creating Hive tables	8486.33 seconds	± 70.37 seconds	± 103.69 seconds	± 239.17 seconds
	Loading with Unicage	3152.00 seconds	± 4.47 seconds	± 6.58 seconds	± 15.18 seconds
4672 GB	Loading with HDFS	6291.00 seconds	± 7.35 seconds	± 10.83 seconds	± 24.98 seconds
	Creating Hive tables	18267.00 seconds	± 3704.97 seconds	± 5459.34 seconds	± 12592.95 seconds
	Loading with Unicage	6360.00 seconds	± 202.96 seconds	± 266.07 seconds	± 689.85 seconds
9417 GB	Loading with HDFS	12937.33 seconds	± 174.52 seconds	± 257.16 seconds	± 593.18 seconds
	Creating Hive tables	36610.33 seconds	± 8439.40 seconds	± 12435.62 seconds	± 28684.99 seconds
	Loading with Unicage	13029.00 seconds	± 343.64 seconds	± 506.37 seconds	± 1168.03 seconds

Table A.6: Data loading - e-commerce tables structured data set: statistically validated execution times.

Input data set volume	Big data system	Mean sample execution time	Confidence levels		
			90%	95%	99%
43 GB	Hive	152.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
	Spark	91.33 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
	Unicage	45.33 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
86 GB	Hive	266.33 seconds	± 1.94 seconds	± 2.86 seconds	± 6.59 seconds
	Spark	147.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
	Unicage	92.00 seconds	± 0.00 seconds	± 0.00 seconds	± 0.00 seconds
173 GB	Hive	498.00 seconds	± 1.69 seconds	± 2.48 seconds	± 5.73 seconds
	Spark	269.67 seconds	± 2.58 seconds	± 3.80 seconds	± 8.77 seconds
	Unicage	185.00 seconds	± 0.00 seconds	± 0.00 seconds	± 0.00 seconds
345 GB	Hive	978.33 seconds	± 9.29 seconds	± 13.69 seconds	± 31.57 seconds
	Spark	501.33 seconds	± 2.58 seconds	± 3.80 seconds	± 8.77 seconds
	Unicage	370.33 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
698 GB	Hive	2096.67 seconds	± 346.54 seconds	± 510.64 seconds	± 1177.88 seconds
	Spark	946.00 seconds	± 1.69 seconds	± 2.48 seconds	± 5.73 seconds
	Unicage	747.00 seconds	± 1.69 seconds	± 2.48 seconds	± 5.73 seconds
1406 GB	Hive	4597.00 seconds	± 543.55 seconds	± 800.94 seconds	± 1847.50 seconds
	Spark	3218.33 seconds	± 68.16 seconds	± 100.43 seconds	± 231.67 seconds
	Unicage	1504.33 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
2823 GB	Hive	10492.67 seconds	± 108.03 seconds	± 159.18 seconds	± 367.19 seconds
	Spark	6193.00 seconds	± 76.08 seconds	± 112.11 seconds	± 258.60 seconds
	Unicage	3019.67 seconds	± 1.94 seconds	± 2.86 seconds	± 6.59 seconds
5672 GB	Hive	25378.00 seconds	± 340.58 seconds	± 501.85 seconds	± 1157.6 seconds
	Spark	13132.67 seconds	± 458.18 seconds	± 675.14 seconds	± 1557.33 seconds
	Unicage	6065.33 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds

Table A.7: Data processing - *select*: statistically validated execution times.

Input data set volume	Big data system	Mean sample execution time	Confidence levels 90%	95%	99%
68 GB	Hive	793.33 seconds	± 54.89 seconds	± 80.88 seconds	± 186.57 seconds
	Spark	283.00 seconds	± 8.77 seconds	± 12.92 seconds	± 29.80 seconds
	Unicage	5925.00 seconds	± 172.18 seconds	± 253.7 seconds	± 585.22 seconds
139 GB	Hive	1623.00 seconds	± 125.61 seconds	± 185.09 seconds	± 426.95 seconds
	Spark	593.00 seconds	± 6.09 seconds	± 8.97 seconds	± 20.69 seconds
	Unicage	12873.00 seconds	± 511.89 seconds	± 754.28 seconds	± 1739.89 seconds
281 GB	Hive	2894.67 seconds	± 235.02 seconds	± 346.31 seconds	± 798.83 seconds
	Spark	1468.00 seconds	± 16.08 seconds	± 23.70 seconds	± 54.67 seconds
	Unicage		Incorrect output		
565 GB	Hive	5028.00 seconds	± 134.48 seconds	± 198.16 seconds	± 457.09 seconds
	Spark	3350.67 seconds	± 46.23 seconds	± 68.12 seconds	± 157.12 seconds
	Unicage		Not benchmarked		
1145 GB	Hive	9786.67 seconds	± 1278.77 seconds	± 1884.29 seconds	± 4346.46 seconds
	Spark	7278.67 seconds	± 148.47 seconds	± 218.78 seconds	± 504.65 seconds
	Unicage		Not benchmarked		
2320 GB	Hive	18471.00 seconds	± 1374.88 seconds	± 2025.91 seconds	± 4673.13 seconds
	Spark	15042.33 seconds	± 609.86 seconds	± 898.64 seconds	± 2072.87 seconds
	Unicage		Not benchmarked		
4672 GB	Hive	37507.00 seconds	± 1403.73 seconds	± 2068.42 seconds	± 4771.18 seconds
	Spark	30242.33 econds	± 603.92 seconds	± 889.89 seconds	± 2052.70 seconds
	Unicage		Not benchmarked		
9417 GB	Hive	71227.33 seconds	± 4318.3 seconds	± 6363.09 seconds	± 14677.62 seconds
	Spark	60130.33 seconds	± 1859.51 seconds	± 2740.03 seconds	± 6320.37 seconds
	Unicage		Not benchmarked		

Table A.8: Data processing - *join*: statistically validated execution times.

Input data set volume	Big data system	Mean sample execution time	Confidence levels 90%	95%	99%
43 GB	Hive	499.67 seconds	± 18.02 seconds	± 26.56 seconds	± 61.25 seconds
	Spark	121.67 seconds	± 2.58 seconds	± 3.80 seconds	± 8.77 seconds
	Unicage	80.33 seconds	± 1.94 seconds	± 2.86 seconds	± 6.59 seconds
86 GB	Hive	978.33 seconds	± 49.33 seconds	± 72.69 seconds	± 167.66 seconds
	Spark	205.33 seconds	± 1.94 seconds	± 2.86 seconds	± 6.59 seconds
	Unicage	166.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
173 GB	Hive	2163.33 seconds	± 103.39 seconds	± 152.35 seconds	± 351.43 seconds
	Spark	374.33 seconds	± 3.51 seconds	± 5.17 seconds	± 11.92 seconds
	Unicage	347.67 seconds	± 0.98 seconds	± 1.44 seconds	± 3.32 seconds
347 GB	Hive	3866.67 seconds	± 159.38 seconds	± 234.85 seconds	± 541.72 seconds
	Spark	702.67 seconds	± 11.48 seconds	± 16.92 seconds	± 39.02 seconds
	Unicage	927.33 seconds	± 3.89 seconds	± 5.74 seconds	± 13.24 seconds
698 GB	Hive	7991.67 seconds	± 290.00 seconds	± 427.32 seconds	± 985.69 seconds
	Spark	1368.00 seconds	± 7.72 seconds	± 11.38 seconds	± 26.24 seconds
	Unicage	1971.33 seconds	± 1.94 seconds	± 2.86 seconds	± 6.59 seconds
1406 GB	Hive	15959.67 seconds	± 1004.08 seconds	± 1479.53 seconds	± 3412.80 seconds
	Spark	3897.33 seconds	± 67.47 seconds	± 99.42 seconds	± 229.32 seconds
	Unicage	4054.33 seconds	± 10.17 seconds	± 14.98 seconds	± 34.55 seconds
2823 GB	Hive	30007.33 seconds	± 1152.30 seconds	± 1697.93 seconds	± 3916.59 seconds
	Spark	7813.33 seconds	± 112.14 seconds	± 165.24 seconds	± 381.17 seconds
	Unicage	8351.00 seconds	± 144.36 seconds	± 212.72 seconds	± 490.67 seconds
5672 GB	Hive		Unable to finish (failed MapReduce stage)		
	Spark	15700.00 seconds	± 83.47 seconds	± 122.99 seconds	± 283.70 seconds
	Unicage	17068.67 seconds	± 50.59 seconds	± 74.55 seconds	± 171.96 seconds

Table A.9: Data processing - *aggregation*: statistically validated execution times.